Learn Programming in Python with Cody Jackson

Grasp the basics of programming and Python syntax while building real-world applications

Packt
www.packt.com

Cody Jackson

Learn Programming in Python with Cody Jackson

Grasp the basics of programming and Python syntax while building real-world applications

Learn Programming in Python with Cody Jackson

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Aaron Lazar
Acquisition Editor: Chaitanya Nair
Content Development Editor: Anugraha Arunagiri
Technical Editor: Ashi Singh
Copy Editor: Safis Editing
Project Coordinator: Ulhas Kambali
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Tom Scaria
Production Coordinator: Shraddha Falebhai

First published: November 2018

Production reference: 1281118

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78953-194-7

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

About the author

Cody Jackson is a disabled military veteran, the founder of Socius Consulting, an IT and business management consulting company in San Antonio, Texas, and a co-founder of Top Men Technologies with Scott Thompson. He is currently employed at CACI International as the lead ICS/SCADA modeling and simulations engineer. He has been involved in the tech industry since 1994, when he left Gateway Computers to join the Navy as a nuclear chemist and radcon technician. Prior to joining CACI, he worked at ECPI University as a computer information systems adjunct professor. He is a self-taught Python programmer, and is the author of *Learning to Program Using Python* and *Secret Recipes of the Python Ninja*. He holds an Associate in Science degree in electromechanical technology, a Bachelor of Science degree in computer engineering technology, and a Master of Science degree in IT management, as well as numerous IT certifications.

I would like to thank my wife and family for having to put up with another book writing disappearing act for the last four months, Scott Thompson for creating the fuel farm schematic drawing, Christopher De La Rosa for allowing me to use the knowledge from work in this book, and the open source community for providing such great tools to work with.

About the reviewers

Nimesh Kiran Verma has a dual degree in math and computing from IIT Delhi, and has worked with companies such as LinkedIn, Paytm, and ICICI for about five years in software development and data science. He co-founded a micro-lending company, Upwards Fintech, and currently serves as its CTO. He loves coding and has mastered Python and its popular frameworks, including Django and Flask. He extensively leverages Amazon Web Services, design patterns, and SQL and NoSQL databases to build reliable, scalable, and low-latency architectures.

Naveen Verma has completed his bachelor's degree in computer science and is a software developer at Turtlemint insurance company. He has knowledge of C, C+, Python, Django, and Android, and is a Java Oracle certified associate, a technology with which he has built various projects. He is also an active open source contributor. He loves solving brainstorming puzzles and ongoing learning is his focus in life.

To our mom and dad, Nutan Kiran Verma and P R Verma, who made us what we are today and gave us the confidence to pursue our dreams. Thanks also to Prabhat, who motivated us to steal time for this book when, in fact, we were supposed to be spending it with him.

Our thanks also to Ulhas and the entire Packt team. Your support was tremendous.

Preface

Much like programming, this book is a fork from another book series: *Learning to Program Using Python*. I started that series in 2008, while deployed in Iraq. I had just learned Python, but didn't feel comfortable with it, especially as the books I had read didn't really "click" with me.

I wrote that series for two reasons. First, I wanted to give back to the open source community. Second, the best way to learn something is to try and teach it to someone else; since I wasn't comfortable as a programmer, writing a book for others would be one of the best ways to ensure I knew what I was talking about.

I also wanted to write the type of book I would have liked to have read when I was learning: written from a personal perspective, rather than an academic viewpoint, and one that points out tips and traps to be aware of. Hence, every book in the series came from that viewpoint.

This new book expands and improves on that series by providing the basics of Python programming, but also walks through programming a real-world scenario: a fuel storage and transfer simulation. In addition, we will also look at how to add a graphical interface to the original, text-based program.

As an introductory book, some of the information presented here may not be completely accurate from a computer science point of view. Even though I have a degree in computer engineering, I consider myself a self-taught programmer, as the majority of my programming has not been in a professional or academic environment, so I may not know all the nuances of software creation.

In addition, being technically accurate isn't necessary for someone new to programming. I'd rather have the reader understand the concepts discussed so as to create a foundation for future learning, than bore the reader so they lose interest. Information will be provided to the best of my knowledge, but terms and theory may be slightly inaccurate to promote reader comprehension.

I'd also like to note that the term "*nix" is used throughout this book to denote any UNIX-like OS, such as Linux and Berkeley Software Distribution, as these OSes tend to have similar functionality. This can also apply to macOS (to an extent), as it has UNIX underpinnings.

Who this book is for

This book is meant for people new to programming, especially the Python language. While no previous programming knowledge is expected, a little knowledge would be beneficial. In addition, while Python is cross-platform, a knowledge of Linux would be helpful.

What this book covers

`Chapter 1`, *The Fundamentals of Python*, looks at the Python language and how it differs from other languages, and covers how to install and use the interactive Python console, commenting code, running Python programs, and alternative programming shells.

`Chapter 2`, *Data Types and Modules*, covers how Python code is structured, common data types and their methods, and how to import and work with Python modules.

`Chapter 3`, *Logic Control*, discusses conditional tests using `if...else` statements, repetition using loops, and error handling with exceptions.

`Chapter 4`, *Functions and Object-Oriented Programming*, looks at optimizing code reuse with functions and objects. Classes, methods, namespaces, and Python properties are also covered.

`Chapter 5`, *Files and Databases*, discusses file interaction, including reading from and writing to files, retrieving individual lines from files, and serializing files for transfer. In addition, basic database operations are covered using SQLite, and a brief summary of using the SQLAlchemy utility for database access is provided.

`Chapter 6`, *Application Planning*, covers the software development life cycle, development practices and methodologies, identifying project requirements, and the use of software repositories.

`Chapter 7`, *Writing the Imported Program*, focuses on the development of the foundational code; identifying specific project requirements; writing utility functions; simulating liquid storage tanks, valves, and pumps; as well as how name mangling alleviates errors in classes.

`Chapter 8`, *Automated Software Testing*, discusses a variety of techniques for writing unit and functional tests, how to use `pytest` to write automated tests, and what code refactoring is.

`Chapter 9`, *Writing the Fueling Scenario*, is the main focus of the book. It details the specific requirements for creating a simulated fuel farm, how the project directory is structured, writing the component instances and scenario functionality, and testing the simulation.

`Chapter 10`, *Software Post-Production*, looks at the final steps to completing software projects, including documenting code using docstrings, creating user documentation via Sphinx, and reviewing lessons learned for the project.

`Chapter 11`, *Graphical User Interface Planning*, explores GUI development, including GUI functionality, elements, and best practices. It also considers the user environment as it applies to GUI creation, and looks at some of the most popular graphical Python frameworks available.

`Chapter 12`, *Creating a Graphical User Interface*, discusses wireframing the GUI prior to writing the code. It then uses the Kivy framework to write the actual interface, utilizing separate Kivy logic and layout files. It also looks at using manual methods to test GUI functionality.

To get the most out of this book

While no programming experience is required, a knowledge of programming terms will be helpful. Knowledge of Linux is a plus. A basic ability to use command-line instructions and the ability to use text editors are assumed.

Unless otherwise stated, this book uses Python 3.6 for all examples.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learn-Programming-in-Python-with-Cody-Jackson`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781789531947_ColorImages.pdf.`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example:"However, when the `print()` function is called in line 11, the text is printed as it was originally entered."

A block of code is set as follows:

```
tank1 = tank.Tank(
           "Tank 1",
           level=36.0,
           fluid_density=DENSITY,
           spec_gravity=SPEC_GRAVITY,
           outlet_diam=16,
           outlet_slope=0.25
    )
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import sys
  if len(sys.argv) > 1:  # Check if arguments are provided
      entered_value = sys.argv[1:]  # Capture all arguments except program name
```

Any command-line input or output is written as follows:

```
$ python foo.py -f /home/User/Documents
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."

Warnings or important notes appear like this.
Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

The Fundamentals of Python

Python is a programming language that, as of the time of writing, is ranked number four on the the TIOBE programming language popularity index. It's one of the most popular languages taught in college, as well as heavily used in industry. Companies such as Google, Rackspace, Industrial Lights and Magic, D-Link, NASA, and others, as well as the Department of Defense and a large number of hobby projects, rely on Python to get work done.

This chapter will cover the following items:

- What is Python?
- Working with Python
- Commenting Python code
- Launching Python programs
- Using the IPython shell

What is Python?

Python is a programming language widely used in a number of applications, such as machine learning, computer graphics, video games, and shell scripts. Nearly any computer application can be implemented in Python, though there are some areas where Python may not be the best solution, such as low-level programs that have to access hardware. In general, though, Python is a good tool for initial application prototyping. Once the initial design has been clarified with Python, it can be re-implemented in a more appropriate language, or the Python code itself can be revised for better optimization.

Python versions

Two main version lines exist for Python: Python 2 and Python 3. Python 2 is the legacy line (version 2.7.15, at the time of writing); while it is still used for some new projects nowadays, it is predominately seen in old software that either can't or won't be upgraded to the Python 3 line. Python 2.7 is the last major release number for this line; incremental upgrades will be provided to back-port Python 3 features or for security patching, but no major features are written for it.

Python 3 (version 3.7.0, as of this writing) is the main development line, and all new features are added here first. Many features in Python 3 are not available in Python 2, or are renamed, so significant effort must be made to convert one version to another.

The Python tools **2to3** and **3to2** are provided with every Python download to help with this conversion process, but they can only handle simple things, such as changing print statements or automatically renaming built-in functions. Anything beyond that requires a programmer to look at the code and make the changes. As this is a non-trivial process (each line of code must be assessed), it may be easier to simply rewrite the code.

Python, as normally used, is technically called CPython, as it is actually written in C code. Python has bindings for use in non-native Python environments, such as Java (Jython), the .NET framework (IronPython), or microcontrollers (MicroPython). This means that you can write regular Python code and it will be interpreted into the correct byte-code for a particular environment. This way, for example, you can interact with a Java program without having to actually write Java code; the Jython interpreter translates Python into equivalent Java code.

Interpreted versus compiled

Python is classified as a scripting language, because it doesn't require a compiler to generate machine code. It actually uses an interpreter to create byte-code, which is cross-platform, and, therefore, any system that has Python installed should be able to run the code. (There are caveats to this, which will be addressed later in the book.)

Byte-code is common among higher-level languages, such as Java, because it makes it easy to write software that runs in many different environments. Languages that use byte-code have a language-specific virtual machine; that is, the virtual machine's sole purpose is to translate the byte-code into something the host computer's operating system can understand. Any OS that has a language-specific virtual machine can process and use the byte-code, thus making an interpreted programming language system agnostic. The programmer doesn't have to do anything special prior to releasing the software.

Machine code is basically the opposite. It is compiled from the raw source code for a particular computer system; this is more common for low-level languages like C++ and Go. The code is portable between systems, but has to be recompiled for each system; it cannot be run immediately like it can with byte-

code. Thus, a programmer must either generate the compiled code for each target OS, or has to provide the source code so an end user can perform that compilation step.

Compiled languages tend to operate faster than interpreted languages because the code has already been optimized for the environment. The compiler also finds many errors before the code is actually executed (the "runtime"). However, compilers can take minutes or even hours to compile the source code, depending on various factors. When errors occur, the programmer has to fix them and rerun the compiler; this compile-fix-compile process continues until the compiler returns no errors.

Compilers can't identify all errors, so the final product must be tested. If problems are found, the code must be fixed, leading to another round of compile-fix-compile, as the fixes to the runtime errors may introduce new errors during compilation.

Working with interpreted languages can be quicker, as there is no compilation step. The code can be run as often as necessary while fixing errors, so the development process is much faster. For many products, developer time is more important than computer time, so having a programmer who can quickly write a program is more desirable than a program that is quicker to run.

In addition, utilizing interpreted languages also allows software developers to provide a scripting interface to the end user; the user can manipulate the program without having to dive into the source code itself. Referring to the previous Jython example, a program written in Java could allow the user to manipulate the data or the actions performed by writing a simple Jython script, essentially adjusting on the fly how the results are generated. This type of customization is commonly found in video games, such as modding communities.

Dynamic versus static

Python is a dynamic typed language. Many other languages are static typed, such as C/C++ and Java. A static typed language requires the programmer to explicitly tell the computer what type of "thing" each data construct is.

For example, if you were writing a C inventory program and one of the variables was `cost`, you would have to declare cost as a float type, which tells the C compiler that the only data that can be used for that variable must be a floating point number, that is, a number with a decimal point, such as 3.14. If any other data type was assigned to that variable, like an integer or a text string, the compiler would give an error when trying to compile the program. (A programming variable is similar to a math variable; it's just a placeholder for a particular value.)

Python, however, doesn't require this. You simply give your variables names and assign values to them. The interpreter takes care of keeping track of the kinds of objects your program is using. This also means that you can change the size of the values as you develop the program. (For the curious, this is handled by adding metadata to a C construct. Every time the item is used, the metadata is looked at to determine how Python should interact with it, as well as potentially modifying the metadata.)

Say you have another decimal number you need in your program. With a static typed language, you have to decide the memory size the variable can take when you first initialize that variable. A double is a floating point value that can handle a much larger number than a normal float (the actual amount of memory used depends on the operating environment, but a float is typically 32 bits long while a double is 64 bits). If you declare a variable to be a float but later on assign a value that is too big to it, your program can develop errors or be slower than expected; changing it to a double will correct these problems.

With Python, it doesn't matter what type of data a construct is. You simply give it whatever number you want, and Python will take care of manipulating it as needed. It even works for derived values. For example, say you are dividing two numbers. One is a floating point number and one is an integer. Python realizes that it's more accurate to keep track of decimals so it automatically calculates the result as a floating point number. The following code example shows what it would look like in the Python interpreter—floating point and integer division:

```
>>> 6.0 / 2
3.0
>>> 6 / 2.0
3.0
```

As you can see, it doesn't matter which value is the numerator or denominator; Python "sees" that a float is being used and gives the output as a decimal value.

This would be a good time to note one of the differences between Python 2 and Python 3. Python 2 truncates division operations, whereas Python 3 automatically converts to decimal values. The following section offers examples of the two versions.

Python 2 versus Python 3 division

While most Python 2.7 code is compatible with 3.x code, you can see that certain things don't carry over well. For example, Python 2 truncates the output of division calculations:

```
# Python 2
>>> 7/2
3
```

Python 3, as the following shows, provides the remainder when dividing. This is important to remember, as the code you're writing will break if it uses features or side effects of a particular version but is run on a different version:

```
# Python 3
>>> 7/2
3.5
```

Working with Python

Python can be programmed through an interactive command line (the interpreter), but anything you code won't be saved. Once you close the session it all goes away. To save your program, it's easiest to just type it in a text file and save it (be sure to use the `.py` extension, that is, `foo.py`).

To use the interpreter, simply type `python` at the command prompt (*nix and Mac) or click the Python application icon (Windows and Mac). If you're using Windows and installed the Python `.msi` file, you should be able to also type `python` at the command prompt, or find the launch icon in the Start menu.

Though they may look the same, the main difference between the Python interpreter and the system command prompt is that the command prompt is part of the operating system while the interpreter is part of Python. The command prompt can be used for other tasks besides messing with Python; the interpreter can only be used for Python.

Note that ***nix** *is used throughout this book to denote any UNIX-like operating system, such as Linux, BSD, and others, as these OSes tend to have similar functionality.*

Installation

Depending on your operating system, Python may already be installed. Python is very prevalent in the *nix world, though different operating systems use different versions. It is almost guaranteed that Python 2 is installed, and an increasing number of systems have some version of Python 3 installed as well.

It is recommended to go to `https://www.python.org` and download the latest version of Python. While this book will focus on version 3.6 and later, the majority of the information will apply to older versions of Python 3 as well. Various installers are available for the major operating systems, as well as some specialized and older platforms; installation instructions are provided with the download.

Launching the Python interpreter

If you're using Linux, BSD, or another *nix operating system, I'll assume you already know about the Terminal; you probably even know how to get Python up and running already. For those who aren't familiar with opening Terminal or the command prompt (same thing, different name on different operating systems), the following sections explain how to do it.

The interactive Python interpreter is sometimes referred to as the Python shell, Python prompt, Python terminal, or Python command line. They all mean the same thing—a special, text-based interface that allows for **Read-Evaluate-Print Loop** (**REPL**) interaction with Python.

Windows (Win8 and above)

The following steps will help you to install Python in Windows:

1. Press the Windows key.
2. Type `cmd` and press *Enter*.
3. You should now have a black window with white text. This is the command prompt.
4. If you type `python` at the prompt, you should be dropped into the Python interpreter prompt. If not, Python isn't installed correctly.

Mac

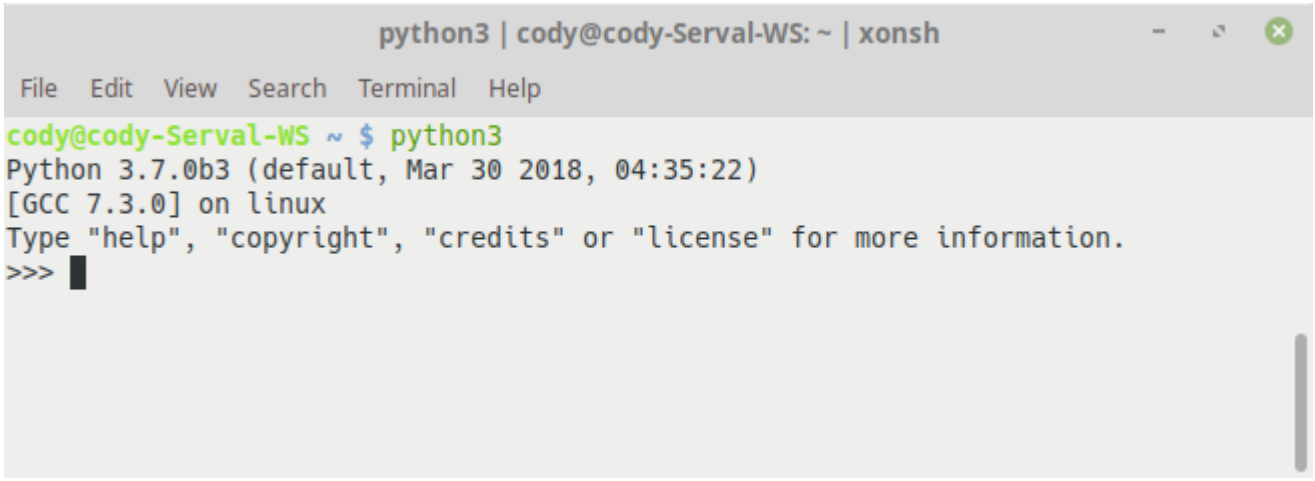
The following steps will help you to install Python in Mac:

1. Open Applications.
2. Open Utilities.
3. Scroll down and open Terminal.
4. You should now have a black window with white text. This is the command prompt.
5. Type `python` at the prompt and you will be in the Python interpreter.

Using the Python command prompt

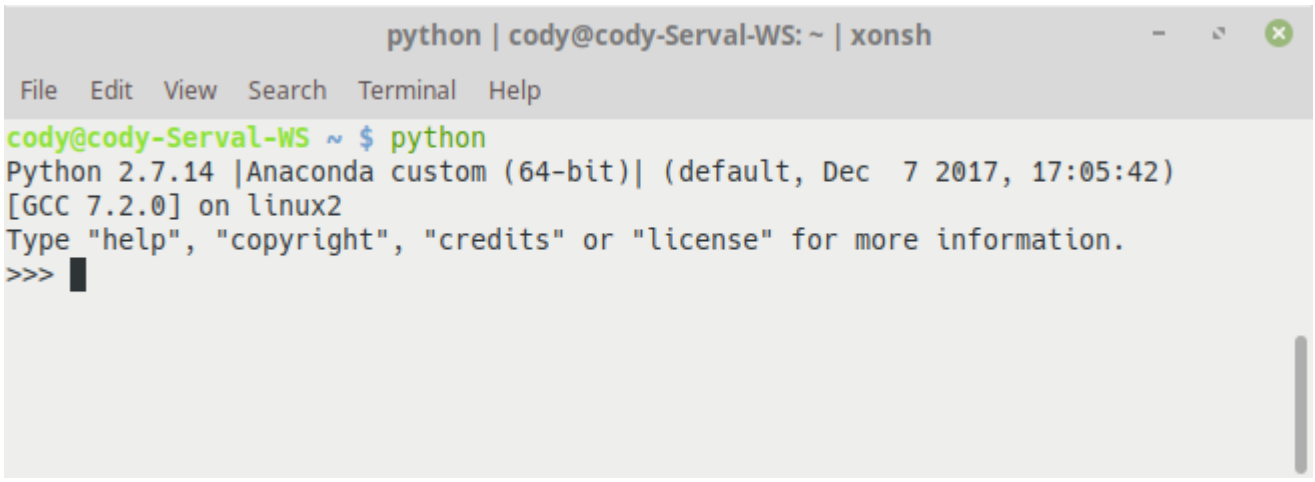
Your Terminal should look similar to the screenshot labeled *Python 3 prompt*. Notice that the command to launch the interpreter is actually `python3`. If both Python 2 and Python 3 are installed on the system, you need to expressly indicate which version to use; otherwise, the system default will be used, which may be different from what is desired. The screenshot labeled *Default Python prompt* shows what the default prompt on the author's system looks like.

Anaconda is a customized Python distribution (`https://www.anaconda.com`) that includes a large number of data science and machine learning tools by default, making it easier for users to manage their environment. This simply demonstrates that, in addition to multiple Python versions, different Python distributions can be installed on the same system, each one customized to a particular use. Below is an example of the interactive Python shell for a vanilla Python installation:


python3 | cody@cody-Serval-WS: ~ | xonsh
File Edit View Search Terminal Help
cody@cody-Serval-WS ~ $ python3
Python 3.7.0b3 (default, Mar 30 2018, 04:35:22)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>

Python 3 prompt

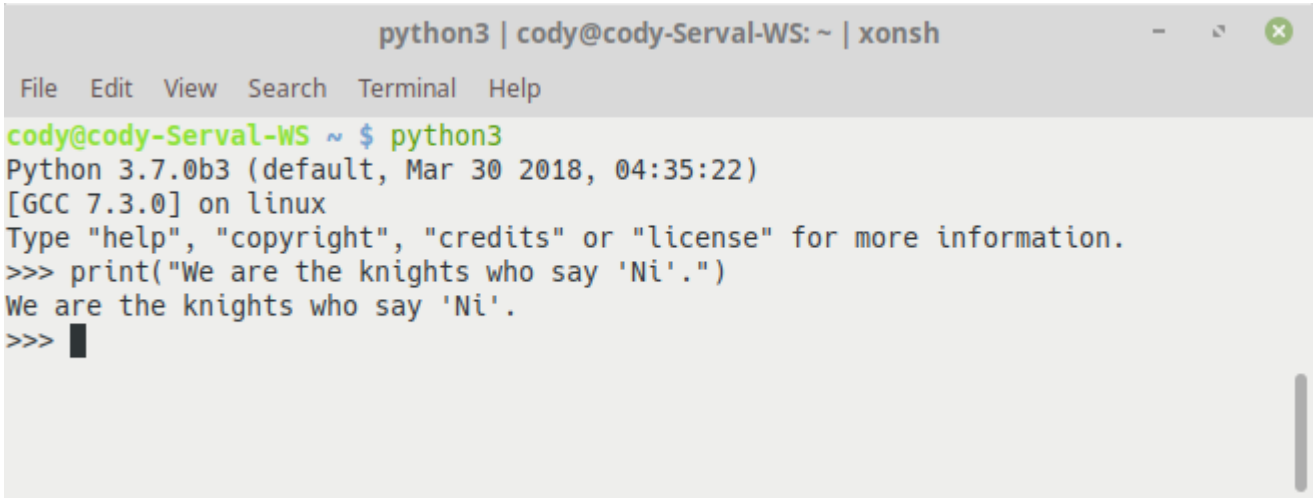
In addition, the astute reader will see there is a difference in the Python environment between the different versions. The following screenshot states that the Python 3 environment is standard Python, whereas the preceding screenshot shows that Python 2 is part of the Anaconda distribution:


python | cody@cody-Serval-WS: ~ | xonsh
File Edit View Search Terminal Help
cody@cody-Serval-WS ~ $ python
Python 2.7.14 |Anaconda custom (64-bit)| (default, Dec 7 2017, 17:05:42)
[GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>

Default Python prompt

For the most part, you won't even notice which version is in use; for example, version 3.4 versus 3.7, unless you are using a library, function, or method for a specific version. Then, you can simply add a special code to identify what version the user has and provide notification to upgrade, or you can modify your code so it is backwards-compatible.

The `>>>` characters in the preceding screenshot is the Python command prompt; your code is typed here and the result is printed on the following line, without a prompt. For example, the following screenshot shows how the user can interact with the Python interpreter just like using a normal operating system command prompt.

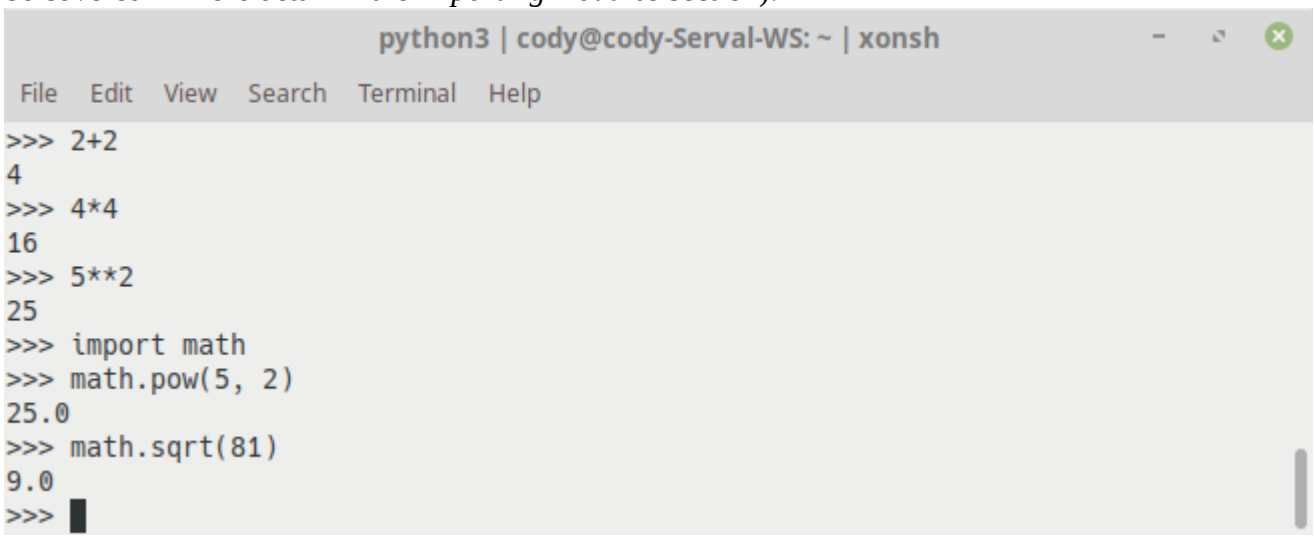

python3 | cody@cody-Serval-WS: ~ | xonsh
File Edit View Search Terminal Help
cody@cody-Serval-WS ~ $ python3
Python 3.7.0b3 (default, Mar 30 2018, 04:35:22)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("We are the knights who say 'Ni'.")
We are the knights who say 'Ni'.
>>>
Python prompt example

If you write a statement that doesn't require any processing by Python, it will simply return you to the prompt, awaiting your next order. In the previous example, the `print()` function simply takes the text that is placed in the parentheses and prints it to the screen. Python doesn't have to do anything with this, in terms of performing calculations or anything, so it prints the statement and then waits for a new command.

(By the way, Python was named after *Monty Python*, not the snake. Hence, some of the code you'll find on the internet, and tutorial, and books will have references to *Monty Python*sketches.)

The standard Python interpreter can be used to test ideas before you put them in your code. This is a good way to test the logic required to make a particular function work correctly or see how a conditional loop will work. You can also use the interpreter as a simple calculator; if you import various mathematical libraries, you can perform complex calculations as well.

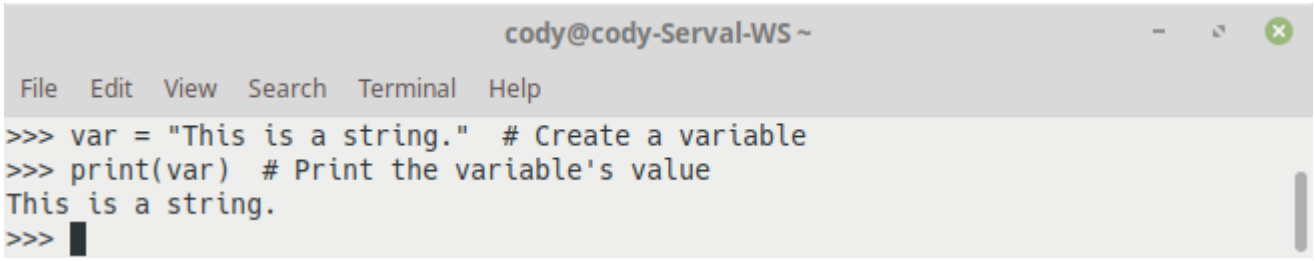
The following screenshot shows the Python interpreter being used for simple arithmetic; it also shows that the math library is imported so more complex calculations can be performed (importing libraries will be covered in more detail in the *Importing modules* section):



Python calculator

Commenting Python code

Another thing to discuss is that comments in Python are marked with the `#` symbol. Comments are used to annotate notes or other information without having Python try to perform an operation on them. For example, the following screenshot demonstrates the use of comments when writing code. It should be noted that, normally, comments in the interactive Python prompt are not used, since it is more of a scratchpad for testing bits of code:


cody@cody-Serval-WS ~
File Edit View Search Terminal Help
>>> var = "This is a string." # Create a variable
>>> print(var) # Print the variable's value
This is a string.
>>>

Python comments

You will see later on that, even though Python is a very readable language, it still helps to put comments in your code. Sometimes, it's to explicitly state what the code is doing, to explain a neat shortcut you used, or to simply remind yourself of something while you're coding, like a "to do" list.

Launching Python programs

If you want to run a Python program, simply type `python foo.py` at the shell command prompt (make sure it's not Python's interactive prompt).

The `foo.py` code is a stand-in term for a generic program; don't try to actually run it because it won't work.

The following screenshot demonstrates how to call a Python program from the command line. This particular program simulates rolling a number of dice; the actual program will be discussed later in this book:

Launching a program

Files saved with the `.py` extension are called modules and can be called individually at the command line or imported into a program, similar to header files in other languages; we saw an example of this in the screenshot labeled *Python calculator*. If your program is going to import other modules, it is easiest to ensure they are all saved in the same directory on the computer, or you have to do some extra work to point to a different directory. More information on working with modules can be found in `Chapter 2`, *Data Types and Modules*, in the *Importing modules* section, or in the Python documentation.

Depending on the program, certain arguments can be added to the command line when launching the program. This is similar to adding switches to a Windows command prompt. The arguments tell the program what exactly it should do.

For example, perhaps you have a Python program that can output its processed data to a file rather than to the screen. To invoke this function in the program you simply launch the program like the following example—launching a Python program with arguments:

```
$ python foo.py -f /home/User/Documents
```

The `-f` argument is received by the program and calls a function that saves the data to the designated location (`/home/User/Documents`) within the computer's filesystem instead of printing it to the screen.

Using the IPython shell

The default Python shell is fine, but there are alternatives. The most popular option is to install the IPython shell from `https://ipython.org`. This is also included with the Anaconda distribution, as well as a number of supporting tools that enhance the development experience.

IPython provides a number of enhancements to the regular interactive Python experience, such as:

- Syntax-highlighted interactive shells
- Web-based notebooks that support multimedia output
- Interactive visualizations

- Interactive parallel application development
- Tab completion
- Object exploration
- Magic functions
- Command history
- Direct implementation of shell commands

The following screenshot demonstrates how the IPython shell differs from the default Python shell. The first thing that is most noticeable is that there is now color within the Python commands. Keywords, errors, and so on are all shown with different colors, easily highlighting different parts of the code.

In addition, each line has its own line number associated with it, rather than the >>> symbol. Lines one and five show that, when necessary, IPython will provide an associated output result if the input command requires it.

Lines six and seven show how IPython can call Bash shell commands directly, in this case pinging a website and printing the current directory, respectively. Because of this ability, some programmers treat IPython as an alternative to the default command shell on *nix systems:

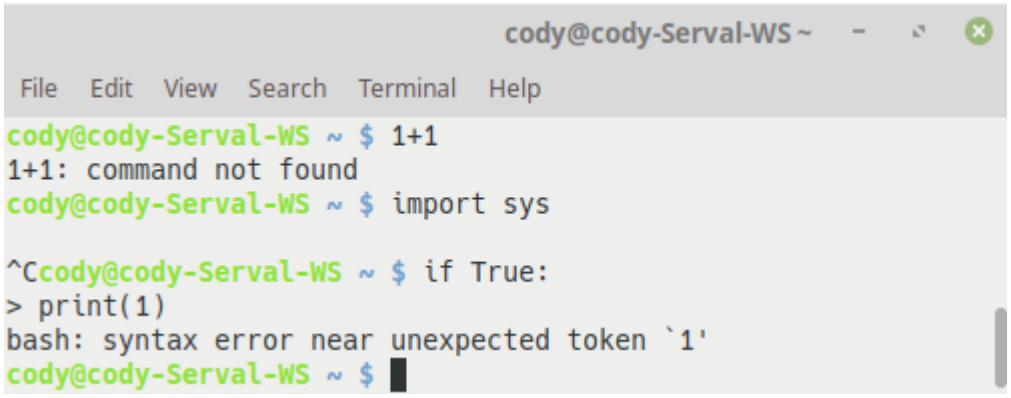
IPython: home/cody
File Edit View Search Terminal Help
cody@cody-Serval-WS ~ $ ipython3
Python 3.6.5 (default, Apr 7 2018, 19:35:51)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.3.1 -- An enhanced Interactive Python. Type '?' for help.
In [1]: 2+2
Out[1]: 4
In [2]: print("I'm a lumberjack")
I'm a lumberjack
In [3]: import math
In [4]: math.exp(2, 8)
TypeError Traceback (most recent call last)
<ipython-input-4-78bd99106cd0> in <module>()
----> 1 math.exp(2, 8)
TypeError: exp() takes exactly one argument (2 given)
In [5]: math.pow(2, 8)
Out[5]: 256.0
In [6]: !ping www.yahoo.com
PING www.yahoo.com(media-router-fp1.prod1.media.vip.gq1.yahoo.com (2001:4998:c:1023::4)) 56 data bytes
64 bytes from media-router-fp1.prod1.media.vip.gq1.yahoo.com (2001:4998:c:1023::4): icmp_seq=1 ttl=48 time=68.4 ms
64 bytes from media-router-fp1.prod1.media.vip.gq1.yahoo.com (2001:4998:c:1023::4): icmp_seq=2 ttl=48 time=67.5 ms
64 bytes from media-router-fp1.prod1.media.vip.gq1.yahoo.com (2001:4998:c:1023::4): icmp_seq=3 ttl=48 time=74.9 ms
^C
--- www.yahoo.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 67.599/70.309/74.915/3.273 ms
In [7]:
KeyboardInterrupt
In [7]:
In [7]: !pwd
/home/cody
In [8]:
IPython example

However, there is an alternative to using IPython in this manner: the Xonsh shell, found at https://xon.sh. Depending on your preference, it is pronounced using the Greek letter Chi ("X"), to sound like "conch," or with a "Z," to sound like "zonsh."

Xonsh is built on Python 3.4 and includes Bash shell functions; it is designed to improve on perceived problems with Bash, as well as making the lives of Python programmers easier. This is because Xonsh essentially replaces the Bash shell with Python, allowing the use of Python code directly at the command line without having to invoke a Python interactive prompt. It also means that Python code in Xonsh has direct access to the underlying OS processing and filesystems, allowing the user to never have to drop back to Bash to interact with the OS.

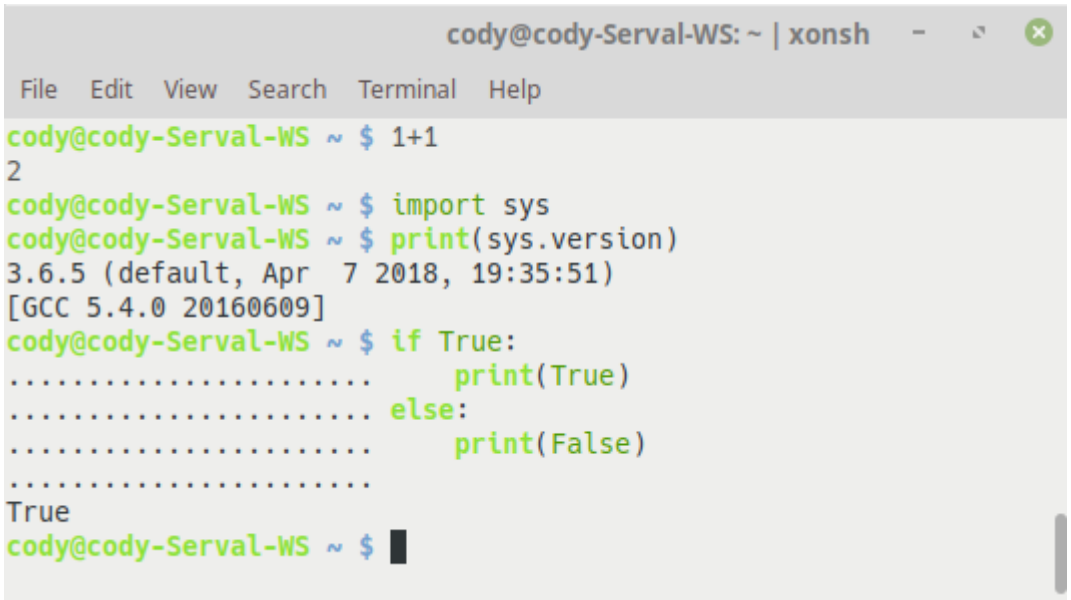
If you look back through the previous screenshots, you'll note that at the top of each window, the name "xonsh" was listed. Xonsh functions just like the default Bash shell in *nix; it's only when you start using commands that are associated with Python that you will notice differences.

The following screenshot shows the errors that occur when trying to run Python commands directly with a Bash shell:


cody@cody-Serval-WS ~
File Edit View Search Terminal Help
cody@cody-Serval-WS ~ $ 1+1
1+1: command not found
cody@cody-Serval-WS ~ $ import sys
^Ccody@cody-Serval-WS ~ $ if True:
> print(1)
bash: syntax error near unexpected token `1'
cody@cody-Serval-WS ~ $

Bash command errors

The following screenshot shows the same commands successfully functioning within the Xonsh shell:



Xonsh commands

For the purposes of this book, normal Bash commands will be used when demonstrating OS shell commands, to limit confusion. However, interactive Python sessions will be demonstrated through IPython, rather than the default Python shell.

Summary

In this chapter, we discussed what the Python programming language is and how it differs from other languages. We learned how to use it on different operating systems and how to interact with the interactive Python shell. We saw how to comment Python code to provide a better explanation of what the code is doing. Finally, we discussed two alternative Python environments: IPython and the Xonsh shell.

In the next chapter, we will take a look at the Python data types (such as lists, dictionaries, and sets), how these types are used when programming, and how Python modules are imported and utilized to enhance coding projects.

Data Types and Modules

Because Python is built upon the C language, many aspects of Python will be familiar to users of C-like languages. However, Python makes life easier because it isn't as low-level as C. The high-level nature of Python means that many data primitives aren't required, as a number of complicated data structures are provided in the language by default.

In addition, Python includes features not often found in low-level languages, such as garbage collection and dynamic memory allocation. On the flip side, Python isn't known for its ability to interact with hardware or perform other low-level work. In other words, Python is great for writing applications but wouldn't be a good choice for writing a graphics card device driver.

Learning how to use built-in data structures helps your programming. Data structures are particular ways of organizing data so they can be used most efficiently. It's easier to write code because the included data structures tend to provide all the features you need, so you spend less time creating your own. If you do need to create your own, you'll probably use the built-in structures as a foundation to start from. This, in turn, means your customized structures will generally perform better than fully customized code, as the built-in data structures have been vetted by multiple developers over a long period of time, so they are fully optimized. Finally, using built-in structures means you always know what is available; proprietary frameworks are an unknown entity, as you can never be sure what is available to you.

In this chapter, we will cover the following topics:

- Structuring code
- Common data types
- Python numbers
- Strings
- Lists
- Dictionaries
- Tuples
- Sets
- Using data type methods
- Importing modules

Structuring code

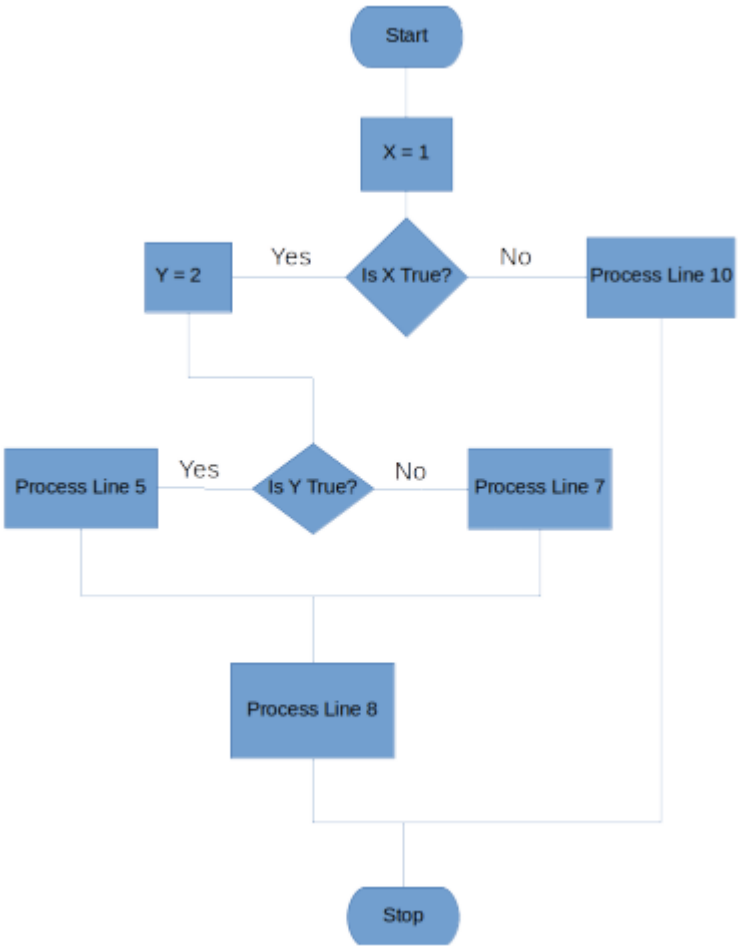
Before we get too far into this chapter, we really need to cover the most obvious and special feature of Python: indentation. Python forces the user to program in a structured format. Code blocks are determined by the amount of indentation used; this is frequently referred to as "white space matters". In many other C-based languages, brackets and semicolons are used to show code grouping or end-of-line termination. Python doesn't require those; indentation is used to signify where each code block starts and ends. In this section is an example of how white space works in Python (line numbers are added for clarification). The following code shows how white space is significant:

```
1  x = 1
2  if x:  # if x is True...
3       y = 2  # process this line
4       if y:  # if y is True...
5           print("x = true, y = true")  # process this line
6       else: # if y is False...
7           print("x = true, y = false")  # process this line
8       print("x = true, y = unknown")  # if x is True, process this line as well
9  else: # if x is False...
10      print("x = false")  # process this line
```

Each indented line demarcates a new code block. To walk through the preceding code snippet, line 1 is the start of the main code block. Line 2 is a new code section; if `x` has a value that is `true`, then indented lines below it will be evaluated. In Python, true can be represented by the word `true`, any number other than `0`, and so on. Hence, lines 3 and *4* are in another code section and will be evaluated only if line 2 is `true`.

Line 5 is yet another code section and is only evaluated if `y` is not a `false` value. Line 6 is part of the same code block as lines 3 and 4; it will also be evaluated in the same block as those lines. Line 9 is in the same section as line 2 and is evaluated regardless of what any other indented lines may do; in this case, this line won't do anything because line 2 is `true`.

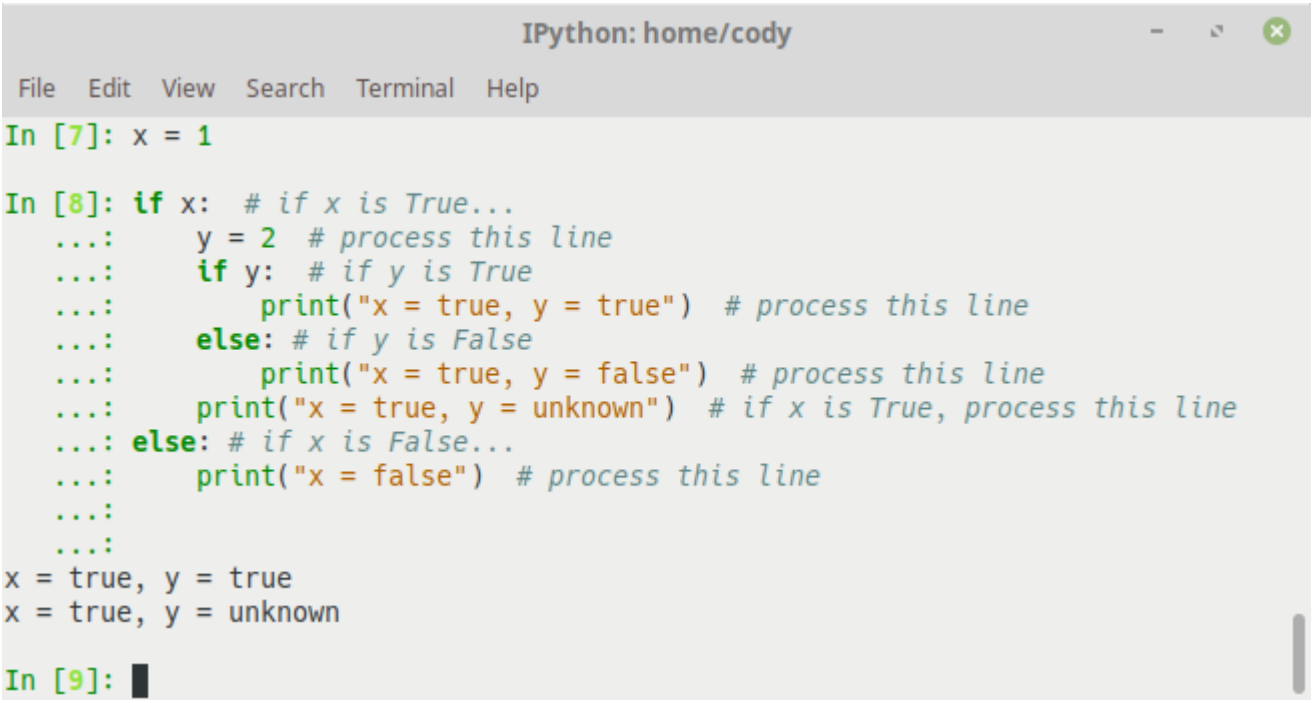
In case that is confusing, the following diagram shows a flowchart of the logic for the previous example. Note that if line 2 is **No** (the first diamond decision icon), the program logic immediately jumps to line 10 and the program ends. Only if variable **X** is **True** will any of the indented lines be evaluated:



Logic flowchart

You'll notice that compound statements, such as the `if` comparisons, are created by having the header line followed by a colon (:). The rest of the statement is indented below it. The biggest thing to remember is that indentation determines grouping; if your code doesn't work for some reason, double-check which statements are indented. Some development environments allow you to toggle vertical lines that make it easier to check indentation.

The following screenshot is an example of running the program in the preceding example. Notice that, since both `x` and `y` are not equal to zero or another false-type value, the if true statements are printed:


IPython: home/cody
File Edit View Search Terminal Help
In [7]: x = 1
In [8]: if x: # if x is True...
...:   y = 2 # process this line
...:   if y: # if y is True
...:   print("x = true, y = true") # process this line
...:   else: # if y is False
...:   print("x = true, y = false") # process this line
...:   print("x = true, y = unknown") # if x is True, process this line
...: else: # if x is False...
...:   print("x = false") # process this line
...:
...:
x = true, y = true
x = true, y = unknown
In [9]:
White space demonstration

Multiple line spanning

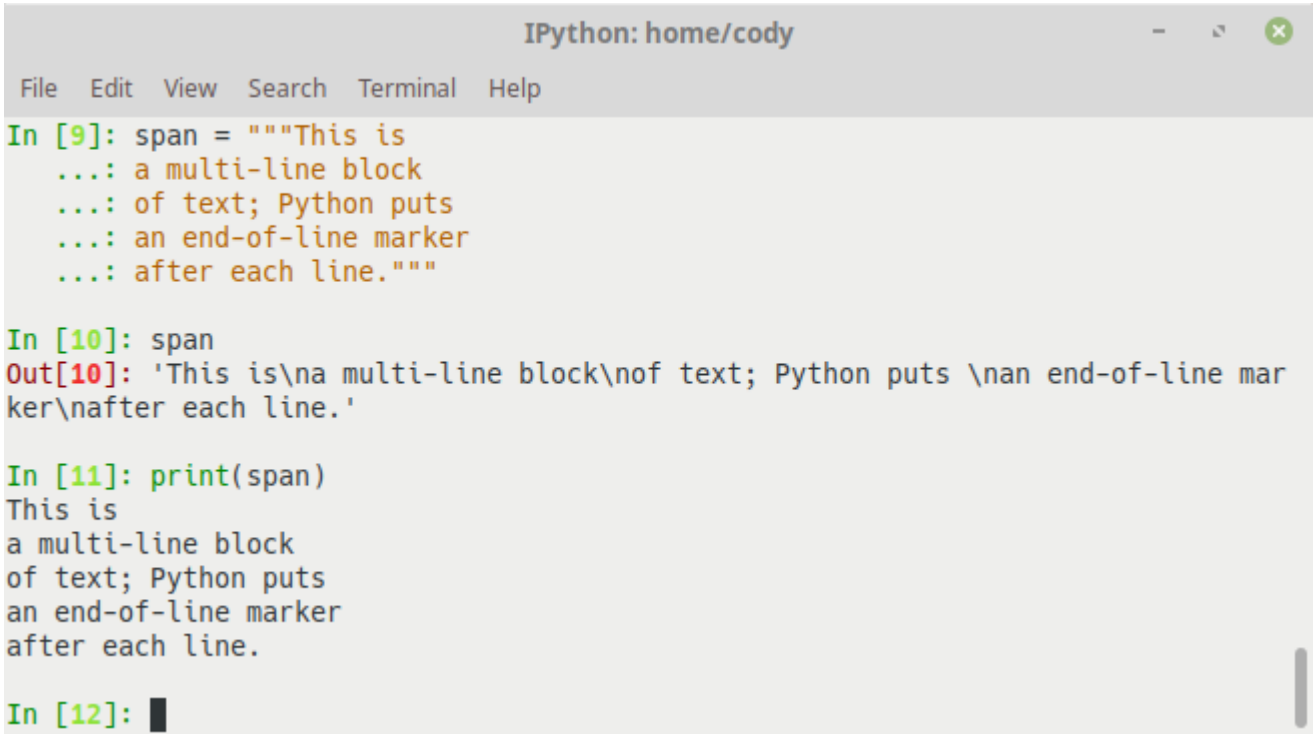
Statements can span more than one line if they are collected within braces (parentheses (), square brackets [], or curly braces {}). Normally parentheses are used. When spanning lines within braces, indentation doesn't matter; the indentation of the initial bracket is used to determine which code section the whole statement belongs to. The following example shows a single variable having to span multiple lines due to the length of the parameters:

```
tank1 = tank.Tank(
            "Tank 1",
            level=36.0,
            fluid_density=DENSITY,
            spec_gravity=SPEC_GRAVITY,
            outlet_diam=16,
            outlet_slope=0.25
    )
```

`tank1` is the variable, and everything to the right of the equal sign is assigned to `tank1`. While Python allows the developer to write everything within the parentheses on one line, the preceding example has separated each parameter into different lines for clarity.

The Python interpreter recognizes that everything within the parentheses is part of the same object (`tank1`), so spreading the parameters across multiple lines doesn't cause a problem.

String statements (text) can also be multiline if you use triple quotes. For example, the following screenshot demonstrates a long block of text that is spread over multiple lines:


IPython: home/cody
File Edit View Search Terminal Help
In [9]: span = """This is
...: a multi-line block
...: of text; Python puts
...: an end-of-line marker
...: after each line."""
In [10]: span
Out[10]: 'This is\na multi-line block\nof text; Python puts \nan end-of-line mar
ker\nafter each line.'
In [11]: print(span)
This is
a multi-line block
of text; Python puts
an end-of-line marker
after each line.
In [12]:
Triple quote line spanning

When the variable containing the text is directly called (line 10 in the preceding screenshot), Python returns the raw text, including the `\n` symbol which represents a newline character; it tells the system where a new line starts and the old one ends. However, when the `print()` function is called in line 11, the text is printed as it was originally entered.

Common data types

Like many other programming languages, Python has built-in data types that the programmer uses to create a program. These data types are the building blocks of the program. Depending on the language, different data types are available. Some languages, notably C and C++, have very primitive types; a lot of programming time is spent simply combining these primitive types into useful data structures.

Python does away with a lot of this tedious work. It already implements a wide range of types and structures, leaving the developer more time to actually create the program. Having to constantly recreate the same data structures for every program is not something to look forward to.

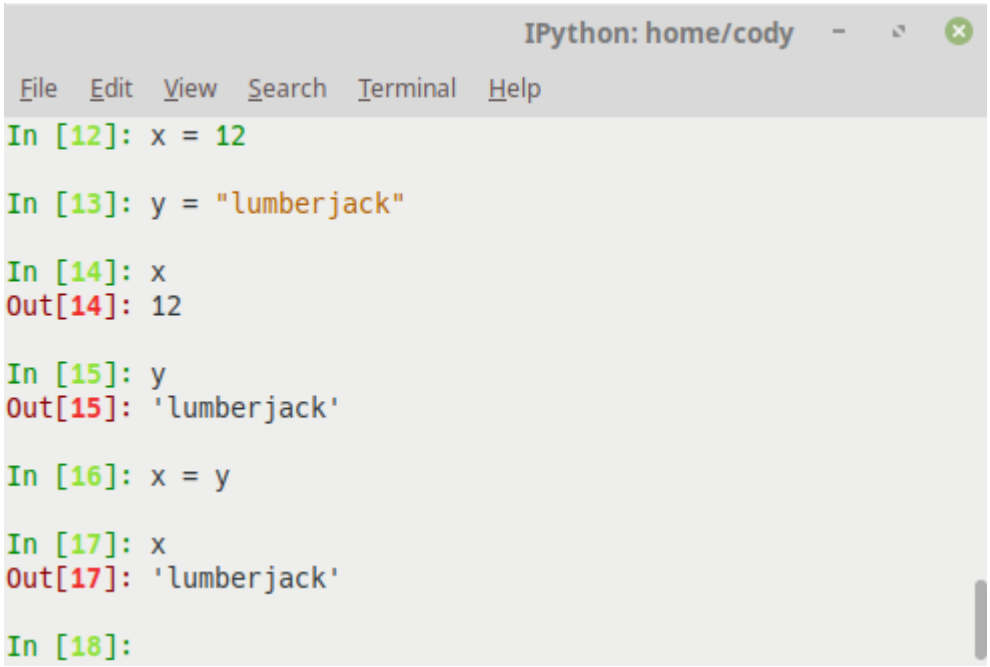
Python has the following built-in types:

- Numbers
- Strings
- Lists
- Dictionaries
- Tuples
- Files
- Sets
- Databases

In addition, functions, modules, classes, and implementation-related types are also considered built-in types, because they can be passed between scripts, stored in other objects, and otherwise treated like the other fundamental types.

Naturally, you can build your own types if needed, but Python was created so that very rarely will you have to roll your own. The built-in types are powerful enough to cover the vast majority of your code and are easily enhanced.

It was mentioned previously that Python is a dynamic typed language; that is, a variable can be used as an integer, a float, a string, or whatever. Python will determine what is needed as it runs. The following screenshot shows how variables can be assigned arbitrarily. You can also see that it is trivial to change a value, as shown in line 16:



Dynamic typing

Other languages often require the programmer to decide what the variable must be when it is initially created. For example, C would require you to declare `x` in the preceding program to be of type `int` and `y` to be of type `string`. From then on, that's all those variables can be, even if, later on, you decide that they should be a different type.

That means you would have to decide what each variable will be when you started your program; that is, deciding whether a number variable should be an integer or a floating-point number. Obviously, you could go back and change them at a later time, but it's just one more thing for you to think about and remember. Plus, any time you forgot what type a variable was and you tried to assign the wrong value to it, you would get a compiler error.

Python also has a difference between expressions and statements. In Python, an expression can contain identifiers (names), literals (constant values of built-in types), and operators (primarily arithmetic-looking symbols, but can be other items, such as [], (), or other symbols). Expressions can be reduced to a derived value, much like solving a math equation.

Statements, on the other hand, are everything that can make up a line (or multiple lines) of code; that is, they actually perform the programming logic. Statements can contain expressions. In other words, lines that equate to a value are expressions, whereas lines that actually do something are statements.

Python numbers

Python can handle normal long integers (the maximum length is determined based on the operating system, just like C), Python long integers (the maximum length is dependent on available memory), floating-point numbers (just like C doubles), octal and hexadecimal numbers, and complex numbers (numbers with an imaginary component).

Here are some examples of these numbers:

- **Integer**: 12345, -32
- **Python integer**: 999999999L (in Python 3.x, all integers are Python integers)
- **Float**: 1.23, 4e5, 3e-4
- **Octal**: 012, 0456
- **Hexadecimal**: 0xf34, 0X12FA
- **Complex**: 3+4j, 2J, 5.0+2.5j

Historically, integers were 16-bit numbers while longs were 32-bit. This could cause problems when using compiled languages, such as C, because trying to store a number that was too big for its data type could cause errors. The largest 16-bit number available is 65535, so trying to store 999999999 in a regular integer would fail.

As advances in computing created 32-bit, then 64-bit, processors and operating systems, the number of bits used has changed, but they are still defined by the OS and the particular programming language compiler used to make the program. In Python 3, all integers are the longest value they can be.

Python has the normal built-in numeric tools you'd expect: expression operators (`*`, `>>`, `+`, `<`, and so on), math functions (`pow`, `abs`, and so on), and utilities (`rand`, `math`, and so on). For heavy number-crunching, Python has the **Numeric Python** (**NumPy**) (`http://www.numpy.org`) extension which has such things as matrix data types. It's heavily used in science and mathematical settings, as its power and ease of use make it equivalent to Mathematica, Maple, and MatLab. It's included with the Anaconda Python distribution, and due to the number of other tools included with Anaconda, that's probably the easiest way to obtain NumPy.

Though this probably doesn't mean much to non-programmers, the expression operators found in C have been included in Python; however, some of them are slightly different. Logic operators are spelled out in Python rather than using symbols; for example, logical AND is represented by the word `and` in Python, not by `&&`, as in C; logical OR is represented by the word `or` in Python, not the double-pipe symbol `||`; and logical NOT uses the word `not` instead of `!`. More information can be found in the Python documentation.

Operator level-of-precedence is the same as C, but using parentheses is highly encouraged to ensure the expression is evaluated correctly and enhances readability. Mixed types (float values combined with integer values) are converted up to the highest type before evaluation; that is, adding a float and an integer will cause the integer to be changed to a float value before the sum is evaluated.

Strings

Strings in programming are simply text; either individual characters, words, phrases, or complete sentences. They are one of the most common elements to use when programming, at least when it comes to interacting with the user. Because they are so common, they are a native data type within Python, meaning they have many powerful capabilities built in. Unlike other languages, you don't have to worry about creating these capabilities yourself.

Strings in Python are different than in most other languages. First off, there are no `char` types, only single character strings (`char` types are single characters, separate from actual strings, used for memory conservation). Strings also can't be changed in-place; a new `string` object is created whenever you want to make changes to it, such as concatenation. This means you have to be aware that you are not manipulating the string in memory; it doesn't get changed or deleted as you work with it. You are simply creating a new string each time.

Empty strings are written as two quotes with nothing in between. The quotes used can be either single or double; my preference is to use double quotes, since you don't have to escape the single quote to use it in a string. That means you can write a statement like the following:

```
"And then he said, 'No way', when I told him."
```

If you want to use just one type of quote mark all the time, you have to use the backslash character to escape the desired quote marks so Python doesn't think it's at the end of the phrase, like this:

```
"And then he said, \"No way\", when I told him."
```

Triple quoted blocks are for strings that span multiple lines, as shown in `Chapter 1`, *The Fundamentals of Python*. Python collects the entire text block into a single string with embedded newline characters. This is good for things like writing short paragraphs of text; for example, instructions, or for formatting your source code for clarification.

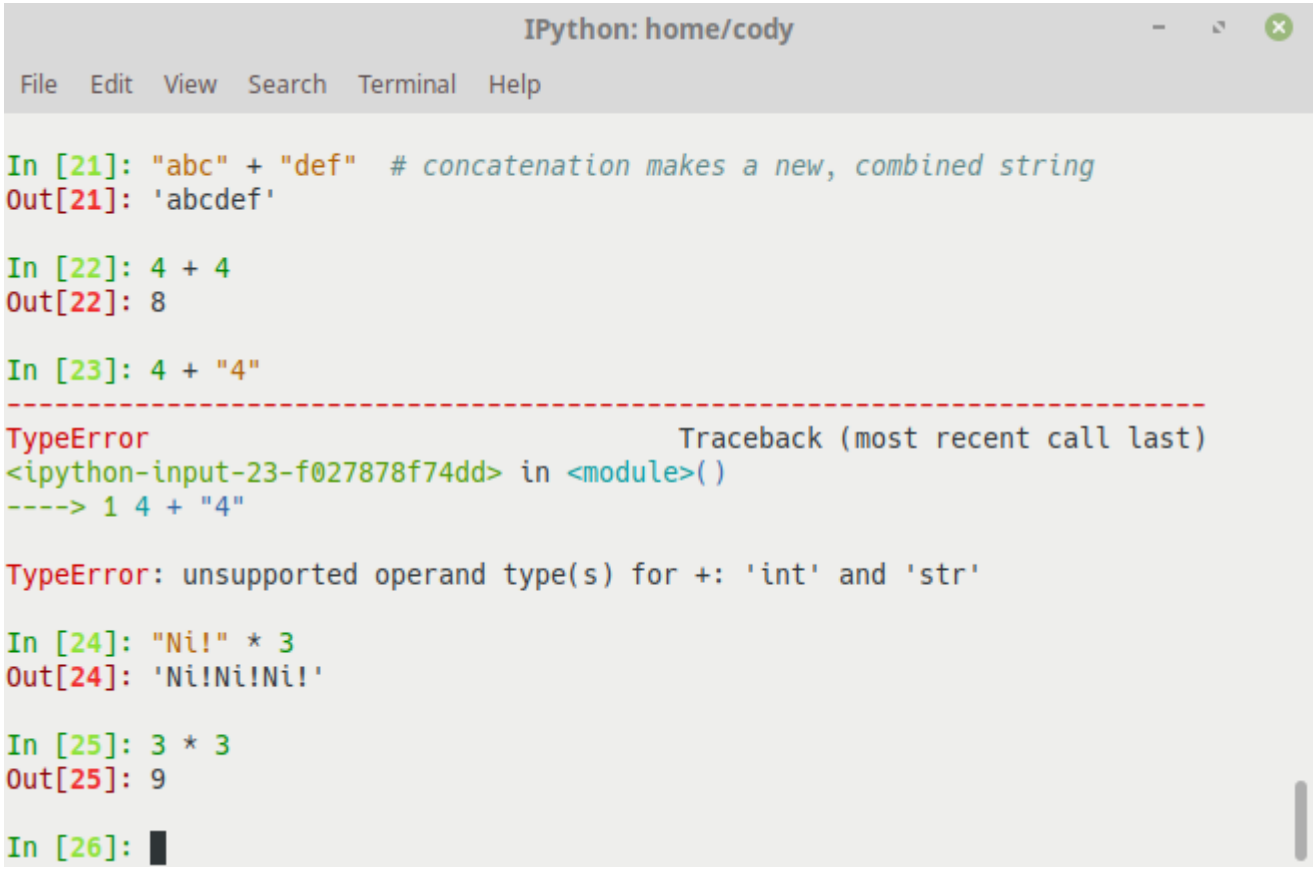
Basic string operations

The + and * operators are overloaded in Python, letting you concatenate (join together) and repeat string objects, respectively. Overloading is just using the same operator to do multiple things, based on the situation where it's used; you'll also see the term polymorphism. For example, the + symbol can mean addition when two numbers are involved or, as in this case, combining strings.

Concatenation combines two (or more) strings into a new string object, whereas repeat simply repeats a given string a given number of times.

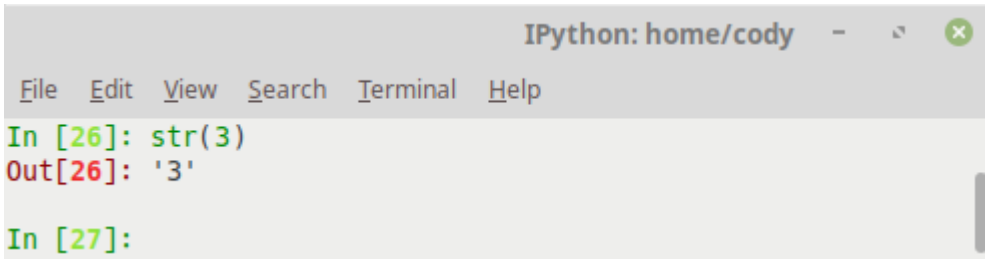
The following screenshot demonstrates some of the Python operators that are overloaded. Line 21 shows string concatenation: the combining of multiple strings into a new, single string. In this case, the + operator is overloaded to combine strings but, when used with numbers (line 22), the operator will provide the sum of the values. Note that in line 23, trying to use the + operator with a number and a string results in an error, as Python doesn't know how to process that command. The error indicates that trying to use the + operator to combine an integer number with a text string won't work, because Python doesn't know whether you want to add two numbers or concatenate two strings. Therefore, the error states that combining the two data types is unsupported.

Line 24 shows the * operator used with a string to return multiple copies of that string in a new, combined string. Line 25 shows normal mathematical use of the * operator, returning the product of two numbers:



Operator overloading

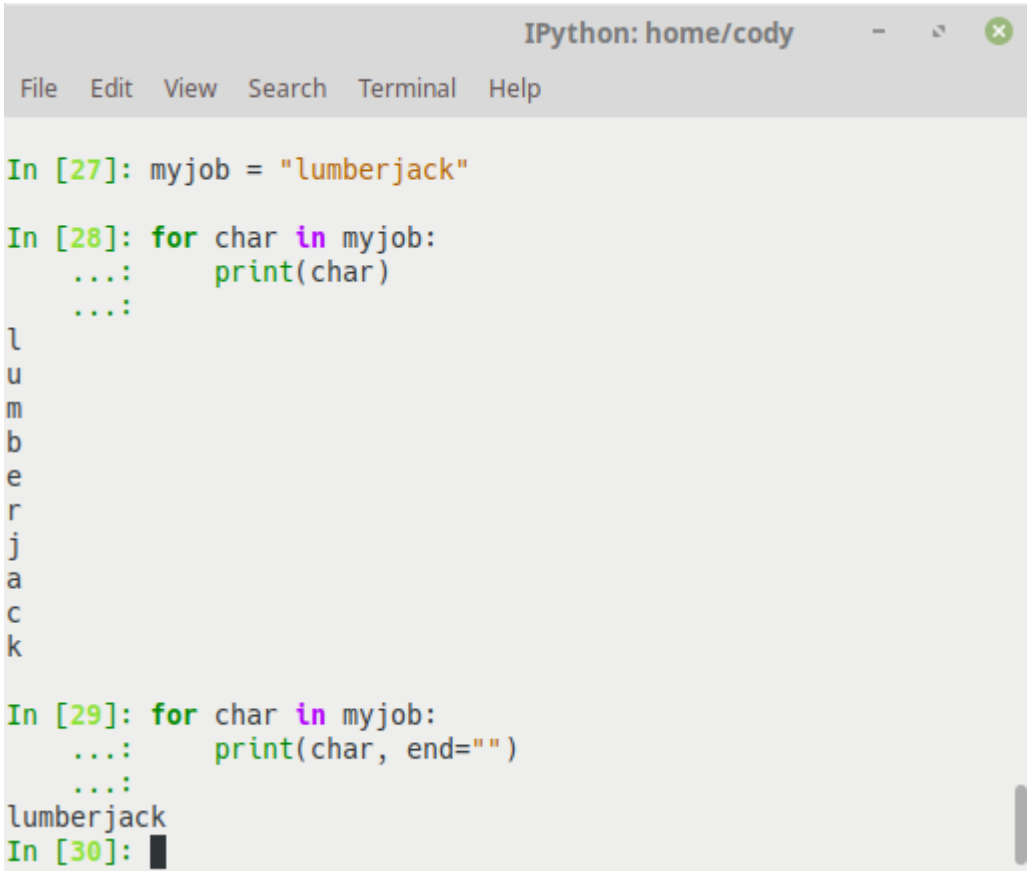
Because Python doesn't know how to combine a string with a number, you can explicitly tell Python that a number should be a string through the `str()` function, as shown in the following screenshot. This is similar to casting values in C/C++. It informs Python that the number is not an integer or floating-point number but is, in reality, a text representation of the number. Just remember that you can no longer perform mathematical functions with it; it's strictly text:



Converting a number to a string

Iteration in strings is a little different than in other languages. Rather than creating a loop to continually go through the string and print out each character, Python has a built-in type for iteration, utilizing a `for` loop. The `for` loops are explained fully in `Chapter 3`, *Logic Control*, in the section, *Loops* but here is a brief explanation: Python accepts a given sequence and then performs one or more actions to each value within the sequence.

The following screenshot provides a demonstration of this:



String iteration

Line 27 assigns a string variable. In line 28, Python is sequentially going through the `myjob`variable and printing each character that exists in the string. By default, the `print()` function assigns a newline

character to the end of each item that is printed. In this case, since we want to print each character that is in the `lumberjack` string, each character will be printed on a separate line.

If you want to print the results on a single line, you'll have to do a little printing manipulation. As a function, `print()` has some additional parameters available; in this case, we can use the `end` keyword as a `print()` argument, as shown in line 29, where the `end` parameter (with no value) has been added. This tells Python that there should be no ending character inserted after printing an individual character, resulting in everything being printed on a single line.

Indexing and slicing strings

Python strings functionally operate the same as Python lists, which are basically C arrays (see the *Lists* section). Unlike C arrays, characters within a string can be accessed both forward and backward. Frontward, a string starts off with a position of `0` and the character desired is found through an offset value (how far to move from the beginning of the string). However, you also can find this character by using a negative offset value from the end of the string. The following screenshot briefly demonstrates this:

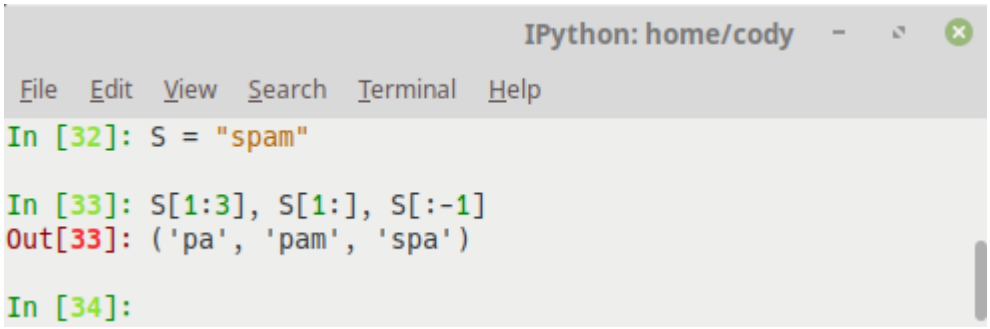
String indexing

Line 30 creates a string variable, and then line 31 requests the characters at position `0` (the very first entry of the string), as well as the second character from the end of the string.

Indexing is simply telling Python where a character can be found within the string. Like many other languages, Python starts counting at `0` instead of `1`. So the first character's index is `0`, the second character's index is `1`, and so on. It's the same counting backward through the string, except that the last letter's index is `-1` instead of `0` (since `0` is already taken). Therefore, to index the final letter, you would use `-1`, the second-to-last letter is `-2`, and so on. Knowing the index of a character is important for slicing.

Slicing a string is basically what it sounds like: by giving upper and lower index values, we can slice the string into sections and pull out just the characters we want. A great example of this is when processing an input file where each line is terminated with a newline character; just slice off the last character and process each line.

The following screenshot demonstrates how string slicing works in more detail.



String slicing

You'll note in the previous screenshot that the colon symbol is used when indicating the slice. The colon acts as a separator between the upper and lower index values. If one of those values is not given, Python interprets that to mean that you want everything from the index value to the end of the string. In the preceding example, the first slice is from index `1` (the second letter, inclusive) to index `3` (the fourth letter, exclusive). You can consider the index to actually be the space before each letter; that's why the letter `m` isn't included in the first slice but the letter `p` is.

The second slice is from index `1` (the second letter) to the end of the string. The third slice starts as the beginning of the string and includes everything except the last character.

One neat feature about the `[:-1]` index: it works on any character, not just letters or numbers. So if you have a newline character (`\n`), you can put `[:-1]` in your code to slice off that character, leaving you with just the text you care about.

You'll see that entering `-1` as the ending value makes it easy to find the end of a string. You could, alternatively, use `len(S)` to get the length of the string, and then use that to identify the last value, but why bother when `[:-1]` does the same thing?

You could also use slicing to process command-line arguments by filtering out the program name. When the Python interpreter receives a program to process, the very first argument provided to the OS is the name of the program. By slicing out the first argument, we can capture the real arguments for processing.

For example, the following code shows how Python can automatically strip out the program's name from a list of arguments passed in to the operating system:

`capture_arguments.py`

```
1  import sys
2  if len(sys.argv) > 1:  # Check if arguments are provided
3      entered_value = sys.argv[1:]  # Capture all arguments except program name
```

String formatting

Formatting strings is simply a way of presenting the information on the screen in a way that conveys the information best. Some examples of formatting are creating column headers, dynamically creating a sentence from a list or stored variable, or stripping extraneous information from the strings, such as excess spaces.

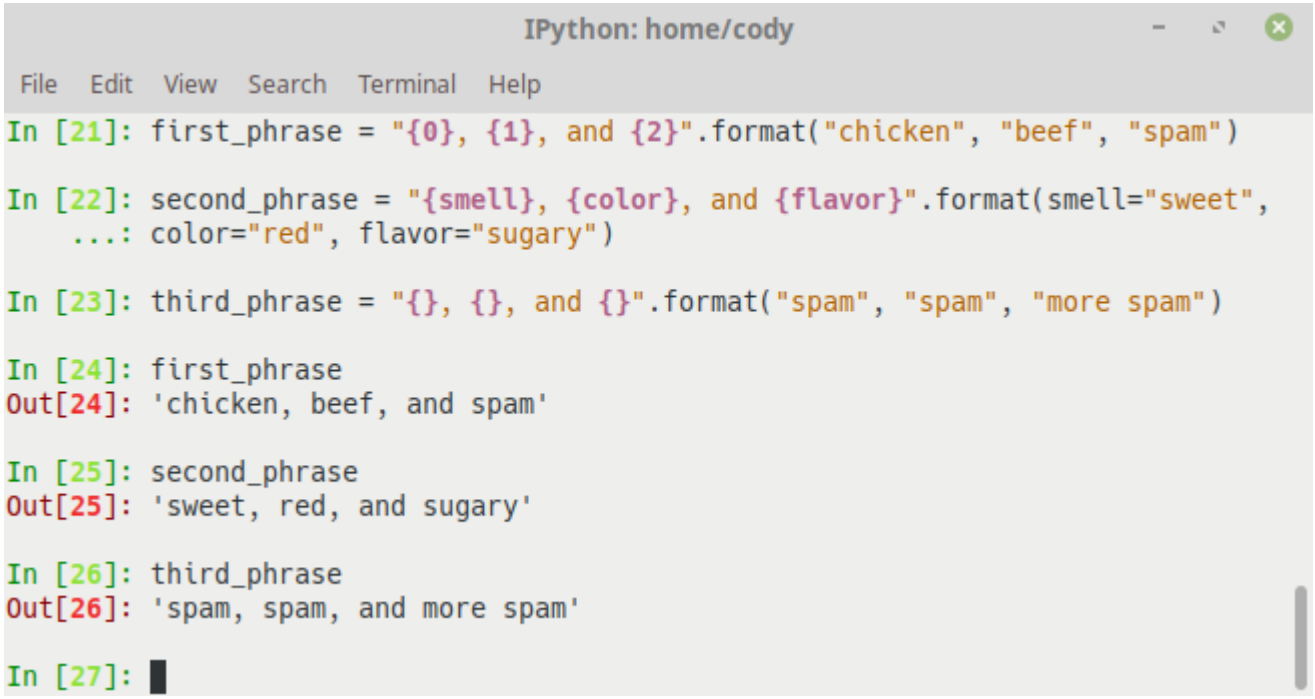
Python supports the creation of dynamic strings. This means you can create a variable containing a value of some type (such as a string or number) and then call that value into your string. You can process a string the same way as in C if you choose to, such as `%d` for integers and `%f` for floating-point numbers.

The following screenshot shows this legacy method of formatting strings. Lines 1 and 2 define the substitution values that will be used. Line 3 creates the string that will be output. Note that the substitution values are identified by position, as well as the fact that, when using an interactive Python session, the interpreter will patiently wait until all required information is presented. In this case, an extra parenthesis is added to complete the `print()` function:

Expression string formatting (pre-Python 2.6)

Since Python 2.6, an alternate way of formatting strings is to use a method call, shown in the following screenshot. Line 21 shows positional substitution, line 22 shows keyword substitution, and line 23 shows relative position substitution. Relative position may be the most common, but it can get out of hand with more than just a few substitutions.

The following screenshot shows an example of method string formatting:


IPython: home/cody
File Edit View Search Terminal Help
In [21]: first_phrase = "{0}, {1}, and {2}".format("chicken", "beef", "spam")
In [22]: second_phrase = "{smell}, {color}, and {flavor}".format(smell="sweet",
...: color="red", flavor="sugary")
In [23]: third_phrase = "{}, {}, and {}".format("spam", "spam", "more spam")
In [24]: first_phrase
Out[24]: 'chicken, beef, and spam'
In [25]: second_phrase
Out[25]: 'sweet, red, and sugary'
In [26]: third_phrase
Out[26]: 'spam, spam, and more spam'
In [27]:
Method string formatting (post-Python 2.6)

Method string formatting is similar to the C# method, but it hasn't replaced the expression formatting. While the expression formatting is deprecated, it is still available in Python 3. However, method formatting provides more capabilities, so expression formatting is primarily found in legacy code.

It's worth pointing out that, while you can frequently get the output you desire by calling the string object directly (as demonstrated in the previous example), it doesn't always work the way you want. Sometimes the object will only return a memory address, particularly if you are trying to print an instance of a class (which will be discussed in `Chapter 4`, *Functions and Object Oriented Programming*, in the *Classes and instances* section). Generally speaking, it's better to explicitly call the `print()` function if you want a statement evaluated and printed out. Otherwise, you don't know exactly what value it will return.

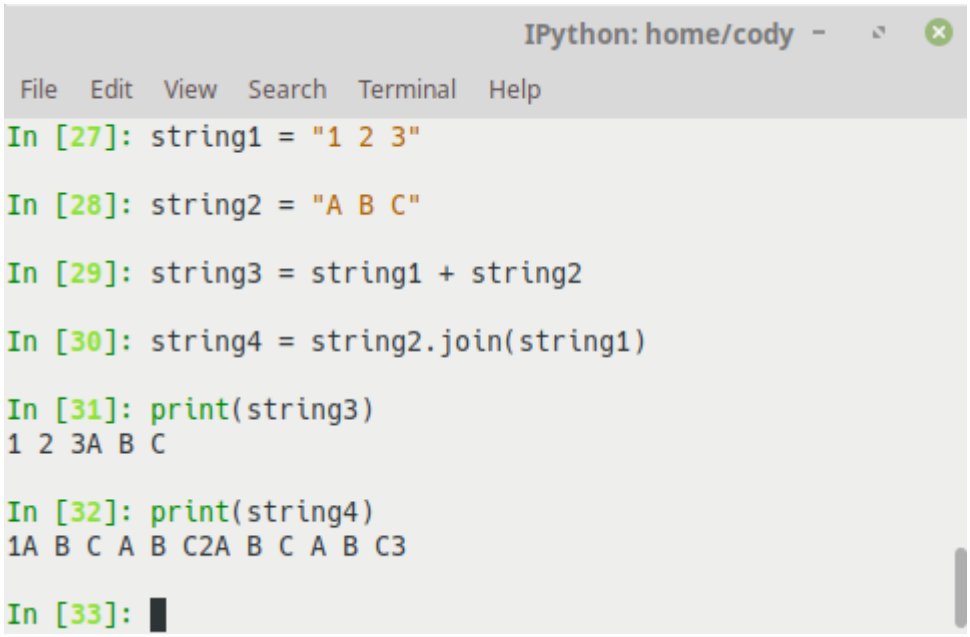
Combining and separating strings

Strings can be combined (joined or concatenated) and separated (split) quite easily. **Tokenization** is the process of splitting something up into individual tokens; in this case, a sentence is split into individual words. When a web page is parsed by a browser, the HTML, JavaScript, and any other code in the page is tokenized and identified as a keyword, operator, variable, and so on. The browser then uses this information to display the web page correctly, or at least as well as it can.

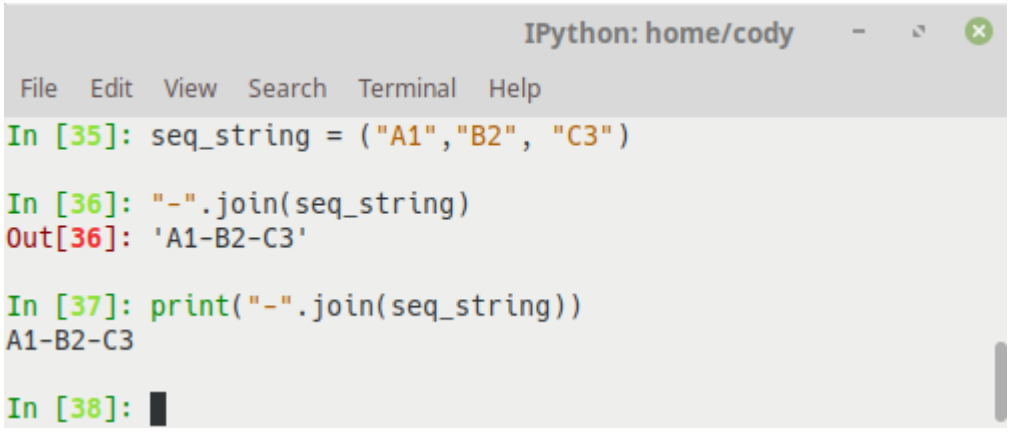
Python does much the same thing. The Python interpreter tokenizes the source code and identifies the parts that are part of the actual programming language and the parts that are data. The individual tokens are separated by delimiters, characters that actually separate one token from another. If you import data into Excel or another spreadsheet program, you will be asked what it should use as a delimiter: a comma, tab, space, and so on. Python does the same thing when it reads the source code.

In strings, the main delimiter is a whitespace character, such as a tab, a newline, or an actual space. These delimiters mark off individual characters or words, sentences, and paragraphs. When special formatting is needed, other delimiters can be specified by the programmer.

String concatenation was demonstrated in *Basic string operations*. An alternative way to combine strings is by joining them. Joining strings combines the separate strings into one string. The catch is that it doesn't concatenate the strings; the `join()` method creates a string in which the elements of a string sequence are joined by a given separator. The following screenshot demonstrates this action. Line 29 is a normal concatenation; the results are printed in line 31. Line 30 joins string `1` with string `2`, with the results in line 32:


IPython: home/cody
File Edit View Search Terminal Help
In [27]: string1 = "1 2 3"
In [28]: string2 = "A B C"
In [29]: string3 = string1 + string2
In [30]: string4 = string2.join(string1)
In [31]: print(string3)
1 2 3A B C
In [32]: print(string4)
1A B C A B C2A B C A B C3
In [33]:
Joining strings

As you can see, the results are not what you expect. The `join()` method is actually designed to be used to create a string where the individual characters are separated by a given separator character. The following screenshot demonstrates this more common use of `join()`:



Common string join

After a sequence of strings is created in line 35 (known as a tuple, and explained further in *Tuples*), the `join()` method is called in two different ways. Line 36 is a simple call of the function itself; the result is a string, with the quotation marks shown. Line 37 is the `print()` function calling `join()`; the resultant string is printed normally, without the quote marks.

Finally, splitting strings separates them into their component parts. The result is a list containing the individual words or characters. The following screenshot shows two ways to split a string:

IPython: home/cody
File Edit View Search Terminal Help
In [38]: split_me = "My wife hates spam."
In [39]: split_me.split()
Out[39]: ['My', 'wife', 'hates', 'spam.']
In [40]: split_me_too = "1, 2, 3"
In [41]: split_me_too.split(",")
Out[41]: ['1', ' 2', ' 3']
In [42]:
Splitting strings

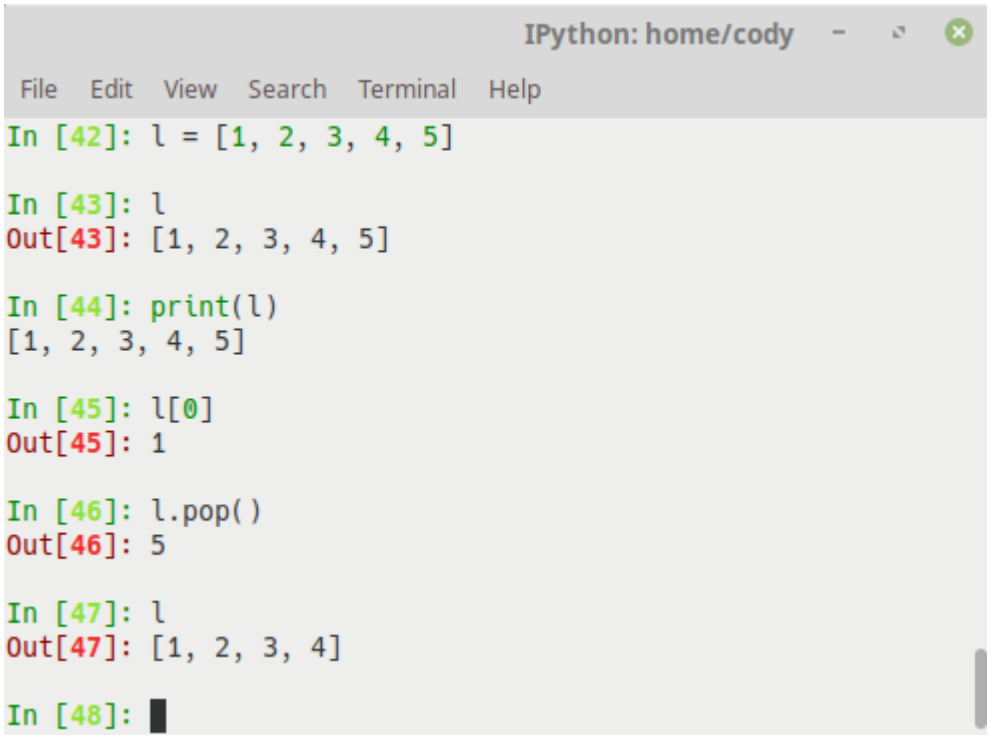
In line 39, the default split is performed, resulting in the string being split at the spaces between words. Line 41 performs the string split on the commas, though essentially any character can be used.

Lists

Lists in Python are one of the most versatile collection object types available. The other workhorses are dictionaries and tuples, but they are really more like variations of lists.

Python lists do the work of most of the data collection structures found in other languages, and since they are built in, you don't have to worry about manually creating them. Lists can be used for any type of object, from numbers and strings to other lists. They are accessed just like strings (since strings are just specialized lists), so they are simple to use. Lists are variable in length; that is, they grow and shrink automatically as they're used, and they can be changed in place; that is, a new list isn't created every time, unlike strings. In reality, Python lists are C arrays inside the Python interpreter and act just like an array of pointers.

The following screenshot shows the creation of a list and a few examples of how to use it:



List examples

After the list is created in line 42, lines 43 and 44 show different ways of getting the values in a list; line 43 returns the list object while line 44 actually prints the items that are in the list. The difference is subtle, but will be more noticeable with more complicated code.

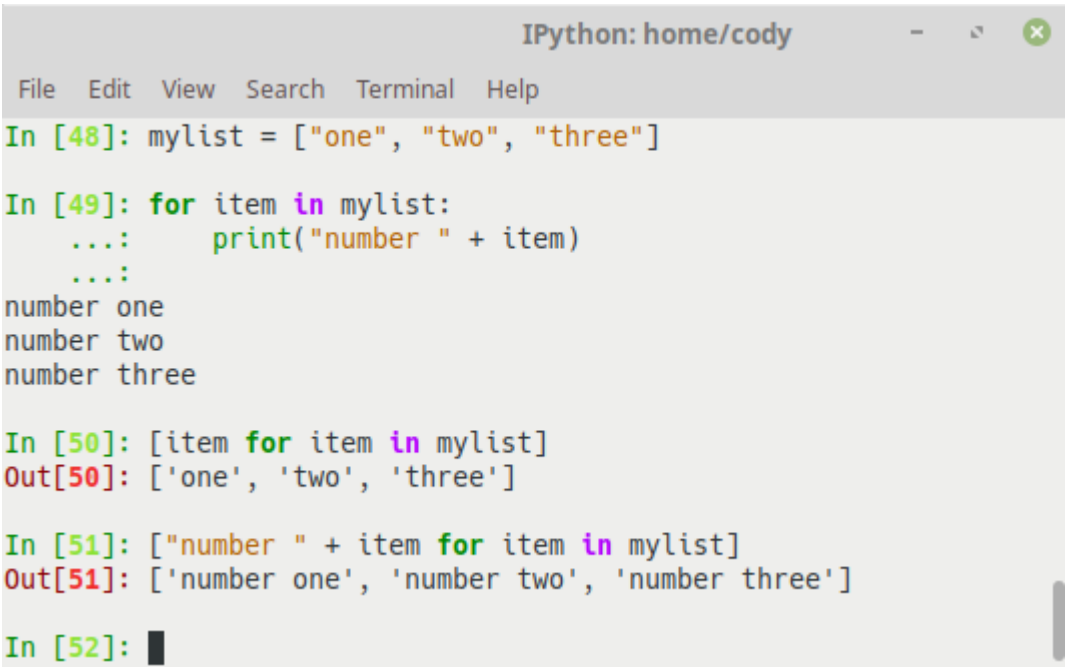
Line 45 returns the first item in the list, while line 46 pops out the last item. Returning an item doesn't modify the list, but popping an item does, as shown in line 47, where the list is visibly shorter.

The biggest thing to remember is that lists are series of objects written inside square brackets, separated by commas. Dictionaries and tuples will look similar except they have different types of brackets.

List usage

Lists are most often used to store homogeneous values; that is, a list usually holds names, numbers, or other sequences that are all one data type. They don't have to; they can be used with whatever data types you want to mix and match. It's just usually easier to think of a list as holding a standard sequence of items.

The most common use of a list is to iterate over the list and perform the same action to each object within the list, hence the use of similar data types. This simple iteration is shown in the following screenshot:


IPython: home/cody
File Edit View Search Terminal Help
In [48]: mylist = ["one", "two", "three"]
In [49]: for item in mylist:
...: print("number " + item)
...:
number one
number two
number three
In [50]: [item for item in mylist]
Out[50]: ['one', 'two', 'three']
In [51]: ["number " + item for item in mylist]
Out[51]: ['number one', 'number two', 'number three']
In [52]:
List iteration

Line 48 defines the list as a sequence of string values. Line 49 creates a `for` loop that iterates through the list, printing out a phrase for each item.

Lines 50 and 51 show alternative ways of iterating through and creating lists. This method is called list comprehension and is frequently found in code as a shortcut to writing a normal `for`loop to make a new list. Line 51 demonstrates that additional information can be provided to the returned values, much like the values returned in line 49.

One thing to note right now, however, is that you can use whatever word for the placeholder that you want; that is, if you wanted to use the name `number` instead of `item` in the preceding examples, you can do that. This is key because it was a weird concept for me when I first encountered it in Python. In other languages, loops like this are either hardwired into the language and you have to use its format or you have to expressly create the `x` value beforehand so you can call it in the loop. Python's way is much easier because you can use whatever name makes the most sense.

Adding list elements

Adding new items to a list is extremely easy. You simply tell the list to add them, as shown in the following screenshot. This also demonstrates how any item can be placed in a list, even disparate data types:

IPython: cody/firefox
File Edit View Search Terminal Help
In [1]: l = [1, 2, 3]
In [2]: l.append("number")
In [3]: l
Out[3]: [1, 2, 3, 'number']
In [4]: l.insert(2, 75)
In [5]: l
Out[5]: [1, 2, 75, 3, 'number']
In [6]:
Appending to a list

The `append()` method simply adds a single item to the end of a list; it's different from concatenation since it takes a single object and not a list. The `append()` method changes the list in-place and doesn't create a brand new list object, nor does it return the modified list. To view the changes, you have to expressly call the list object again, as shown in line 3. So be aware of that in case you are confused about whether the changes actually took place.

If you want to put the new item in a specific position in the list, you have to tell the list which position it should be in; that is, you have to use the index of what the position is. This is demonstrated in line 4 of the previous screenshot.

You can add a second list to an existing one by using the `extend()` method. Essentially, the two lists are concatenated (linked) together, as shown in the following screenshot:

IPython: cody/firefox
File Edit View Search Terminal Help
In [6]: new_l = ["Mary", "had", "a", "little", "spam."]
In [7]: l.extend(new_l)
In [8]: l
Out[8]: [1, 2, 75, 3, 'number', 'Mary', 'had', 'a', 'little', 'spam.']
In [9]:
Extending a list with another list

Be aware that there is a distinct difference between `extend()` and `append()`. The `extend()` function takes a single argument, which is always a list, and adds each of the elements of that list to the original list; the two lists are merged into one. The `append()` function takes one argument, which can be any data type, and simply adds it to the end of the list; you end up with a list that has one element, which is the appended object.

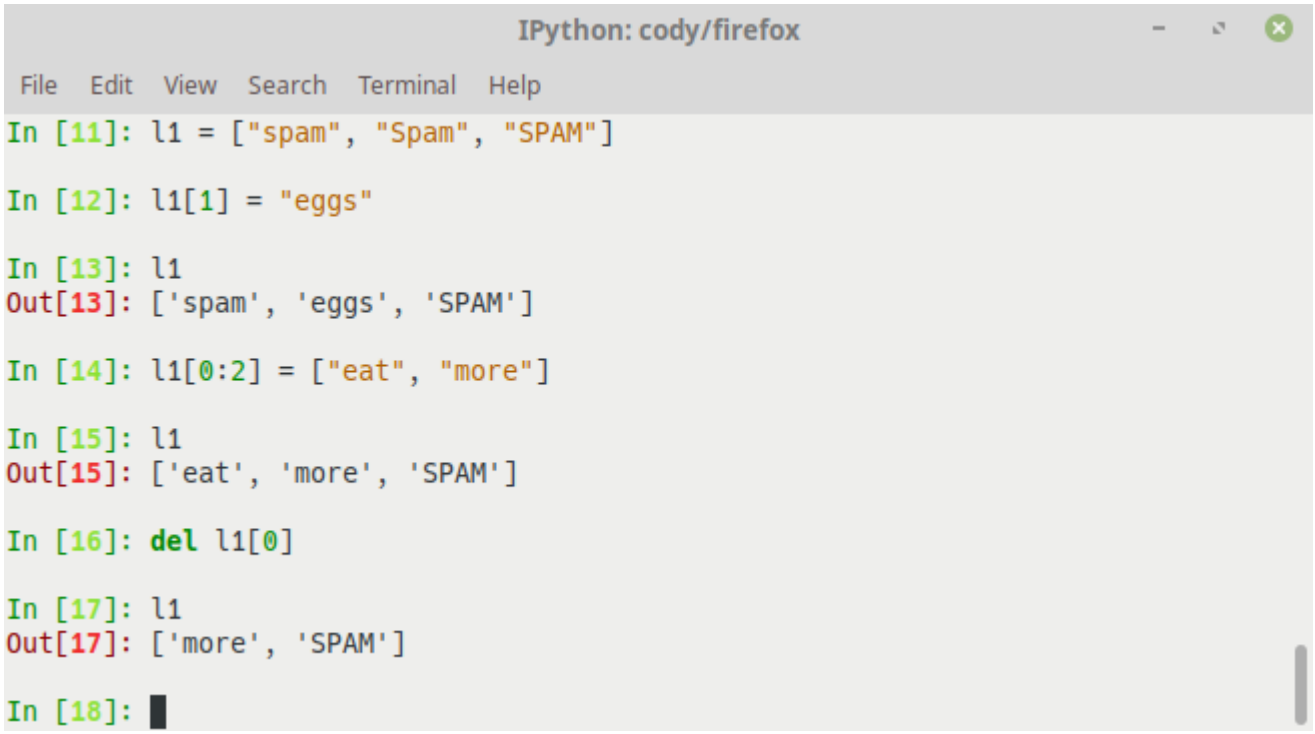
Compare line 10 in the following screenshot to line 8 in the previous screenshot. Whereas appending the `new_l` list to the original list simply added each item from `new_l` to the original, essentially increasing the number of elements, when extending the exact same `new_l` list to the original, the entire list object was added, rather than the individual elements.

extend() versus append()

Mutability

As mentioned several times, one of the special things about lists is that they are mutable; that is, they can be modified in place without creating a new object. The big concern with this is remembering that, if you do this, it can affect other references to it. However, this isn't usually a large problem; it's more of something to keep in mind if you get program errors.

The following screenshot is an example of changing a list using index offset, slicing, and deleting elements:


IPython: cody/firefox
File Edit View Search Terminal Help
In [11]: l1 = ["spam", "Spam", "SPAM"]
In [12]: l1[1] = "eggs"
In [13]: l1
Out[13]: ['spam', 'eggs', 'SPAM']
In [14]: l1[0:2] = ["eat", "more"]
In [15]: l1
Out[15]: ['eat', 'more', 'SPAM']
In [16]: del l1[0]
In [17]: l1
Out[17]: ['more', 'SPAM']
In [18]:
Changing a list

Line 12 changes the value for the element at index `1` (second position in the list). Line 14 swaps out the first two elements for new values. Line 16 deletes the first element; removing multiple elements through slicing is also allowed.

Dictionaries

Next to lists, dictionaries are one of the most useful data types in Python. Python lists, as previously shown, are ordered collections that use a numerical offset. To select an item in a list, you need to know its position within the list. Python dictionaries are unordered collections of objects, matched to a key name; in other words, you can reference an item simply by knowing its associated key.

Because of their construction, dictionaries can replace many typical search algorithms and data structures found in C and related languages. For those coming from other languages, Python dictionaries are just like a hash table or associative array, where an object is mapped to a key name.

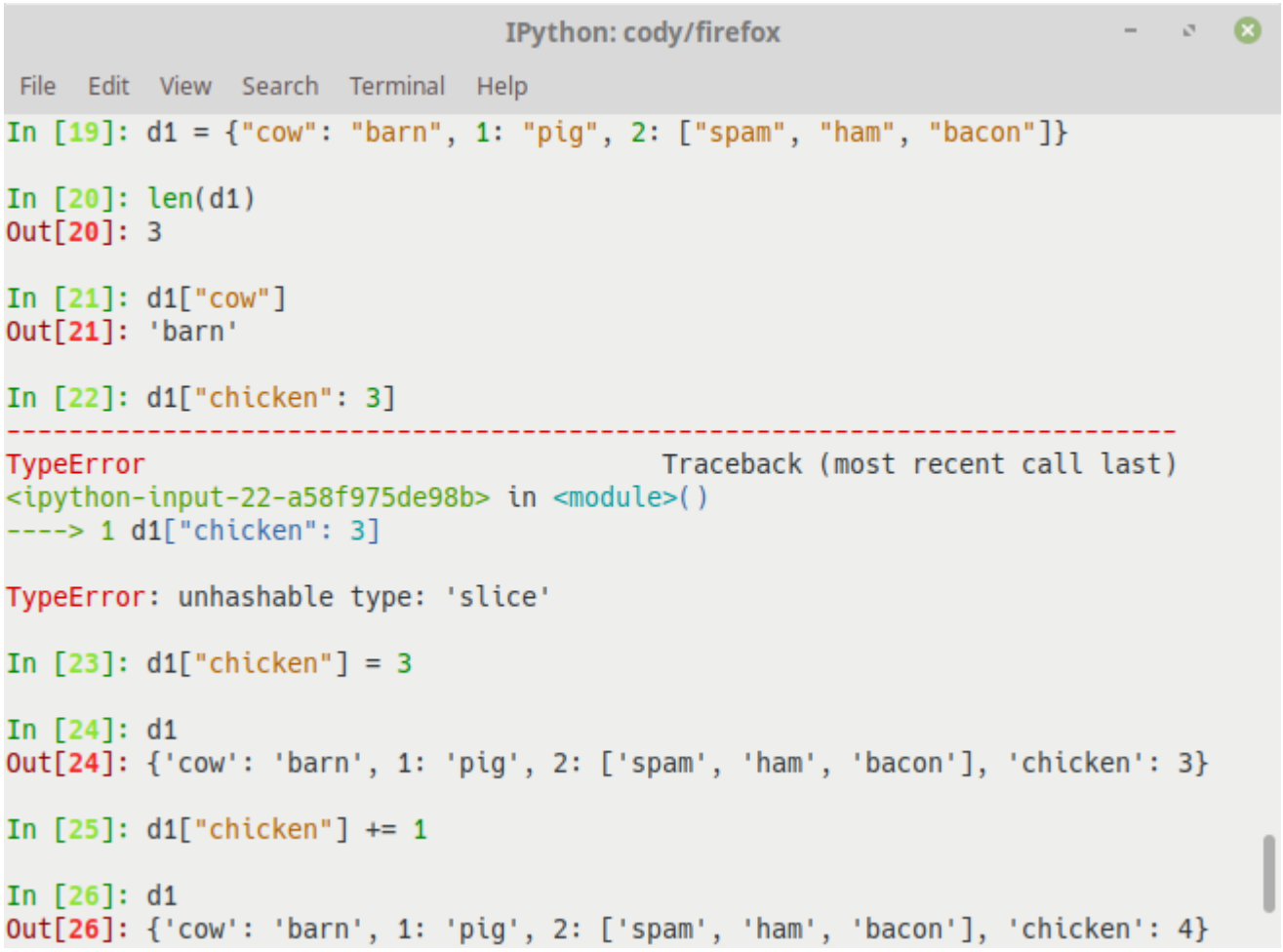
Dictionaries include the following properties:

- They are accessed by a key, not an offset. Each item in the dictionary has a corresponding key; the key is used to call the item.
- Stored objects are in a random order to provide faster lookup. When created, a dictionary stores items in a particular order that makes sense to Python, but may not make sense to the developer.
- To get a value, simply supply the key. If you need to order the items within a dictionary, there is a container called `OrderedDict` that was added in Python 2.7, but it has to be imported from the `collections` library.
- Dictionaries are variable-length, can hold objects of any type (including other dictionaries), and support deep nesting (multiple levels of items can be in a dictionary, such as a list within a dictionary within another dictionary).
- They are mutable but can't be modified like lists or strings. They are the only data type that supports mapping; that is, a key is linked to a value.

Internally, a dictionary is implemented as a hash table.

Creating dictionaries

As previously stated, you create dictionaries and access items through a key. The key can be of any immutable type, such as a string, number, or tuple; basically, anything that can't be changed. Each key's associated value can be any type of object, including other dictionaries. The basic use of dictionaries is displayed in the following screenshot:



Basic dictionary operations

Line 19 creates the dictionary. Note that the brackets for dictionaries are curly braces, the separator between a key word and its associated value is a colon, and that each key:value is separated by a comma. In this example, the first mapping is a string to a string, the second is a string to an integer, and the last is a list to an integer.

Line 20 shows how to see how many items are contained within a dictionary. This value is only the number of mappings, not the individual keys/values contained within the dictionary.

Line 21 returns the value associated with the key `cow`. If you want to add a new item, you have to use the format in line 23—the name of the dictionary, followed by the new key within square brackets, and then what that key is equal to. If you try to make a new dictionary entry by trying to directly map the value to its key through a colon character (line 22), you will get an error.

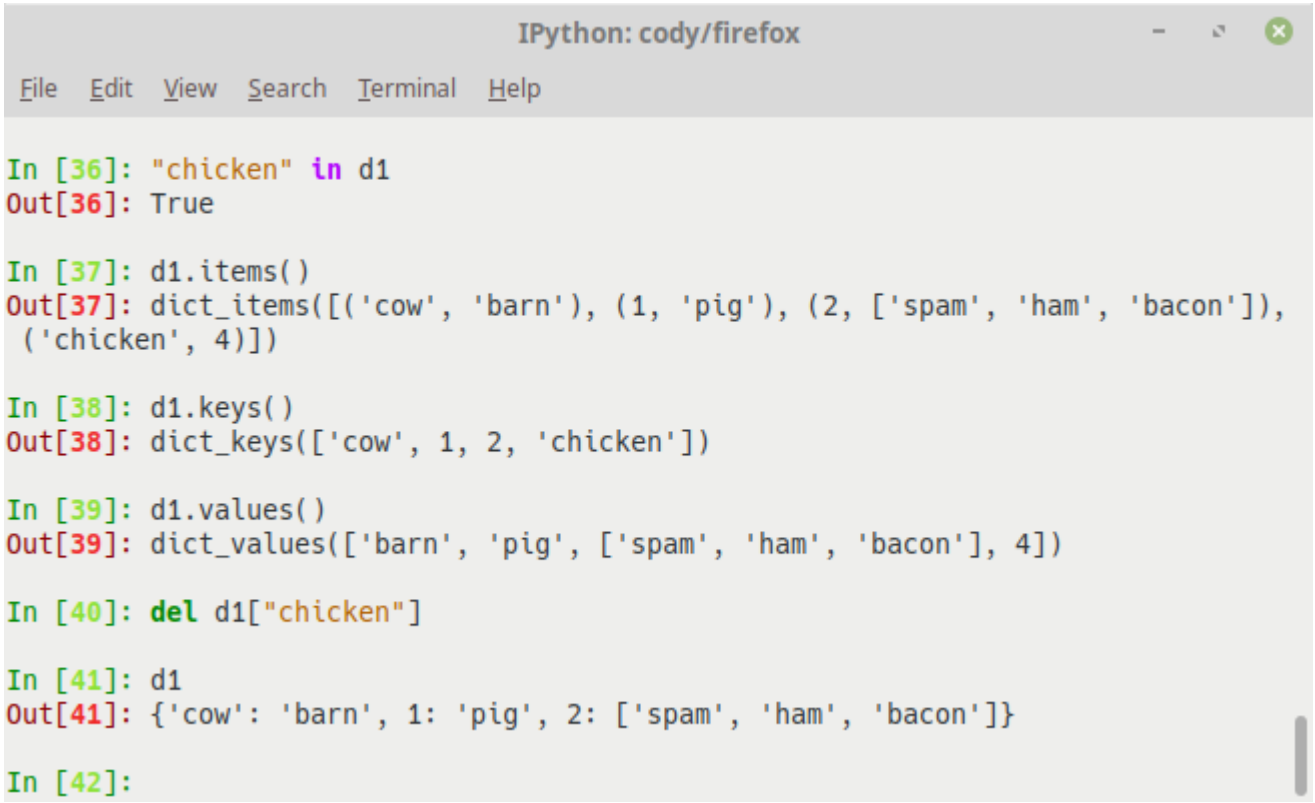
After the new entry is created in line 23, we can verify it is there by simply calling the dictionary (line 24). Values in dictionary entries are completely accessible; in the case of line 25, we can increment the integer value by directly adding `1` to the appropriate key.

Compare this to lines 21 and 23. In line 21, calling the key returned its associated value. In line *23*, adding the `=` sign to a key made a new dictionary entry. Thus, line 25 acts like a combination of those two—it gets the value associated to a key, and then makes a new dictionary entry by performing an operation on the value. In this case, we are simply adding 1 to the value, and then reassigning it as the key's associated value. Line 26 returns the entire dictionary to show that the new value associated with the `chicken` key has been incremented from `3` to `4`.

Working with dictionaries

There are a large number of methods that can be used with dictionaries.

We won't talk about all of them, but some of the more common ones are shown in the following screenshot:


IPython: cody/firefox
File Edit View Search Terminal Help
In [36]: "chicken" in d1
Out[36]: True
In [37]: d1.items()
Out[37]: dict_items([('cow', 'barn'), (1, 'pig'), (2, ['spam', 'ham', 'bacon']),
('chicken', 4)])
In [38]: d1.keys()
Out[38]: dict_keys(['cow', 1, 2, 'chicken'])
In [39]: d1.values()
Out[39]: dict_values(['barn', 'pig', ['spam', 'ham', 'bacon'], 4])
In [40]: del d1["chicken"]
In [41]: d1
Out[41]: {'cow': 'barn', 1: 'pig', 2: ['spam', 'ham', 'bacon']}
In [42]:
Dictionary methods

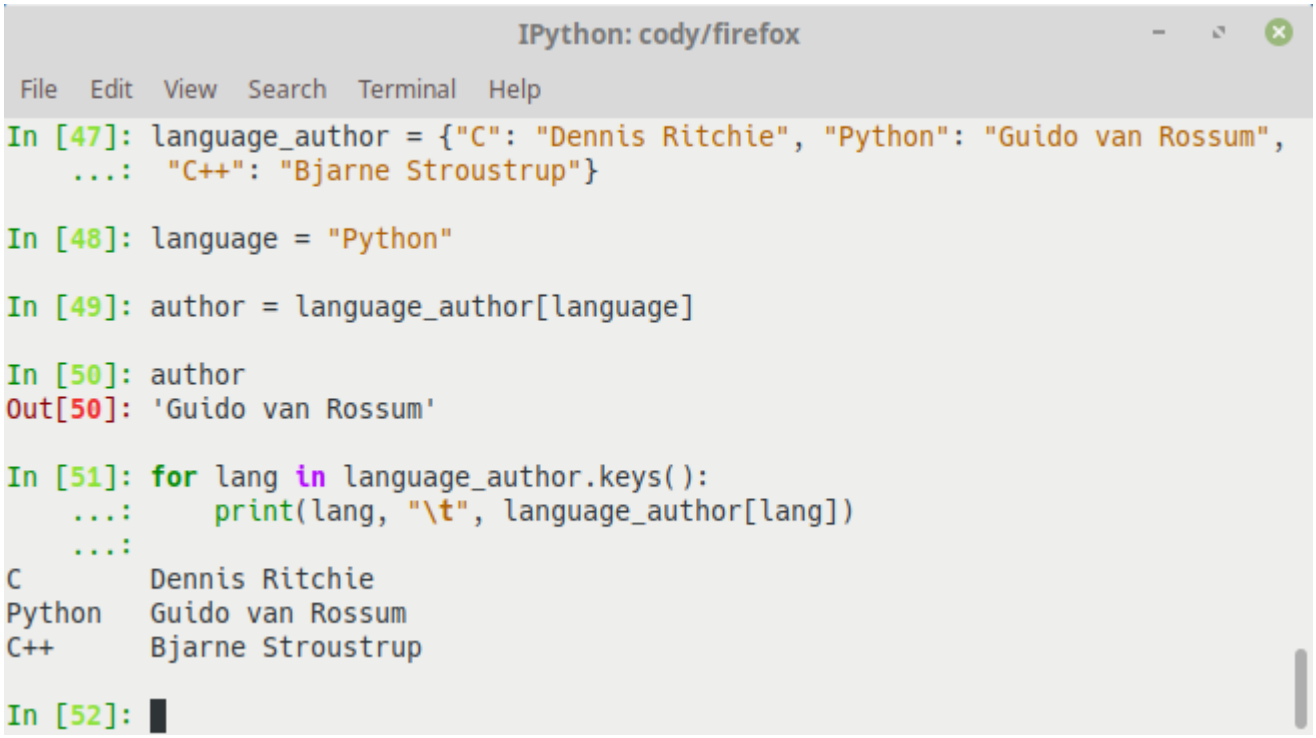
Line 36 checks to see whether a specified key exists within the dictionary.

Line 37 returns all the items that exist within the dictionary—both keys and their associated values. For more flexibility, you can look for just a dictionary's keys (line 38) or just the values (line 39).

To remove entries within a dictionary, you can delete single items, as demonstrated in line 40. To remove all entries in dictionary `d1`, you would use the `d1.clear()` method .

Since dictionaries are changeable, you can add and delete values to them without creating a new dictionary object, as shown in lines 23 and 40. Adding a new object to a dictionary only requires making a new key and value, whereas lists will return an index out-of-bounds error if the offset is past the end of the list. Therefore, you must use `append()` to add values to lists but simply make new key value entries for dictionaries.

The following screenshot is a more realistic dictionary example. The following example creates a table that maps programming language names (the keys) to their creators (the values). You fetch a creator name by indexing on the language name:


IPython: cody/firefox
File Edit View Search Terminal Help
In [47]: language_author = {"C": "Dennis Ritchie", "Python": "Guido van Rossum",
...: "C++": "Bjarne Stroustrup"}
In [48]: language = "Python"
In [49]: author = language_author[language]
In [50]: author
Out[50]: 'Guido van Rossum'
In [51]: for lang in language_author.keys():
...: print(lang, "\t", language_author[lang])
...:
C Dennis Ritchie
Python Guido van Rossum
C++ Bjarne Stroustrup
In [52]:
Using a dictionary

From this example, you might notice that the last command is similar to string and list iteration using the `for` command. However, you'll also notice that, since dictionaries aren't sequences (that is, the stored items are indexed by keyword and not position), you can't use the standard `for` statement. You must use the `keys()` method to return a list of all the keys which you can then iterate through like a normal list.

You may have also noticed that dictionaries can act like light weight databases. The preceding example creates a table, where the programming language column is matched by the creator's row. If you have a need for a database, you might want to consider using a dictionary instead. If the data will fit, you will save yourself a lot of unnecessary coding and reduce the headaches you would get from dealing with a full-blown database. Granted, you don't have the flexibility and power of a true database, but for quick-and-dirty solutions, dictionaries will suffice.

Dictionary details

There are a few key points about dictionaries that you should be aware of:

- Sequence operations don't work. As previously stated, dictionaries are mappings, not sequences. Because there's no order to dictionary items, functions such as concatenation and slicing don't work.
- Assigning new indexes adds entries. Keys can be created when making a dictionary (that is, when you initially create the dictionary) or by adding new values to an existing dictionary. The process is similar and the end result is the same.
- Keys can be anything immutable. The previous examples showed keys as string objects, but any non-mutable object (such as numbers) can be used for a key. Numbers can be used to create a list-like object but without the ordering. Tuples are sometimes used to make compound keys; class instances that are designed not to change can also be used if needed.

Tuples

The final built-in data type is the tuple. Python tuples work exactly like Python lists except they are immutable; that is, they can't be changed in place. They are normally written inside parentheses to distinguish them from lists (which use square brackets), but as you'll see, parentheses aren't always necessary; however, a comma is always required, as expressions can use parentheses too. Since tuples are immutable, their length is fixed. To grow or shrink a tuple, a new tuple must be created.

Since parentheses can surround expressions, you have to show Python when a single item is actually a tuple by placing a comma after the item. A tuple without parentheses can be used when a tuple is unambiguous. However, it's easier to just use parentheses than to screenshot out when they're optional.

Why use tuples?

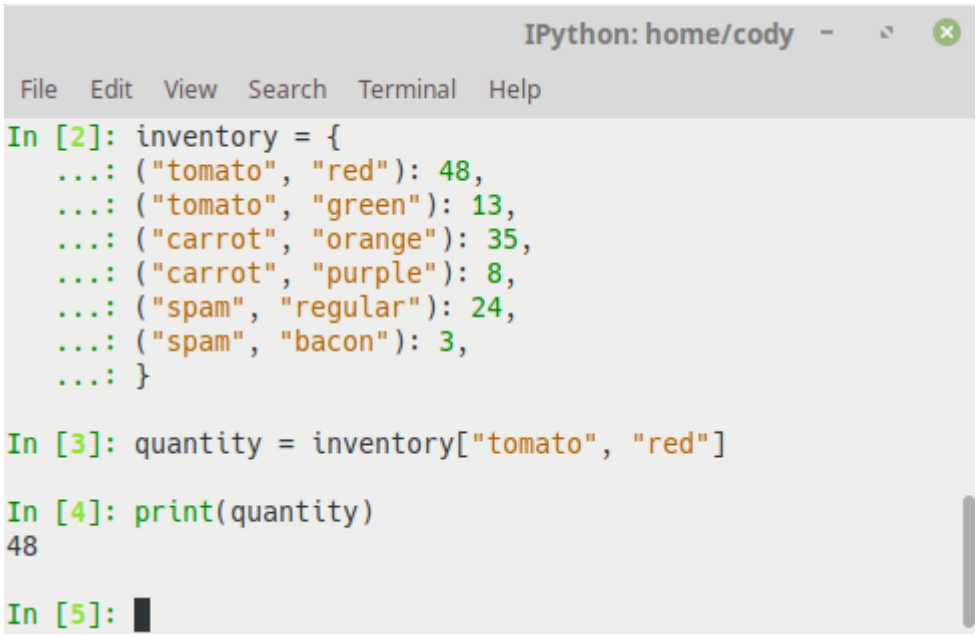
Tuples typically store heterogeneous data, similar to how lists typically hold homogeneous data. It's not a hardcoded rule but simply a convention that some Python programmers follow. Because tuples are immutable, they can be used to store different data about a certain thing. For example, a contact list could conceivably be stored within a tuple; you could have a name and address (both strings) plus a phone number (integer) within a data object.

The biggest thing to remember is that standard operations, such as slicing and iteration, return new tuple objects. Commonly, lists are used for everything except when a developer doesn't want a collection to change. It cuts down on the number of collections to think about; plus, tuples don't let you add new items to them or delete data. You have to make a new tuple in those cases.

There are a few times when you simply have to use a tuple because your code requires it. However, a lot of times you never know exactly what you're going to do with your code and having the flexibility of lists can be useful.

So why use tuples? Apart from sometimes being the only way to make your code work, there are few other reasons to use tuples:

- Tuples are processed faster than lists. If you are creating a constant set of values that won't change, and you need to simply iterate through them, use a tuple.
- The sequences within a tuple are essentially protected from modification. This way, you won't accidentally change the values, nor can someone misuse an API to modify the data. (An API is an application programming interface. It allows programmers to use a program without having to know the details of the whole program.)
- Tuples can be used as keys for dictionaries. One possible use of this is a crude inventory system, such as the following screenshot:

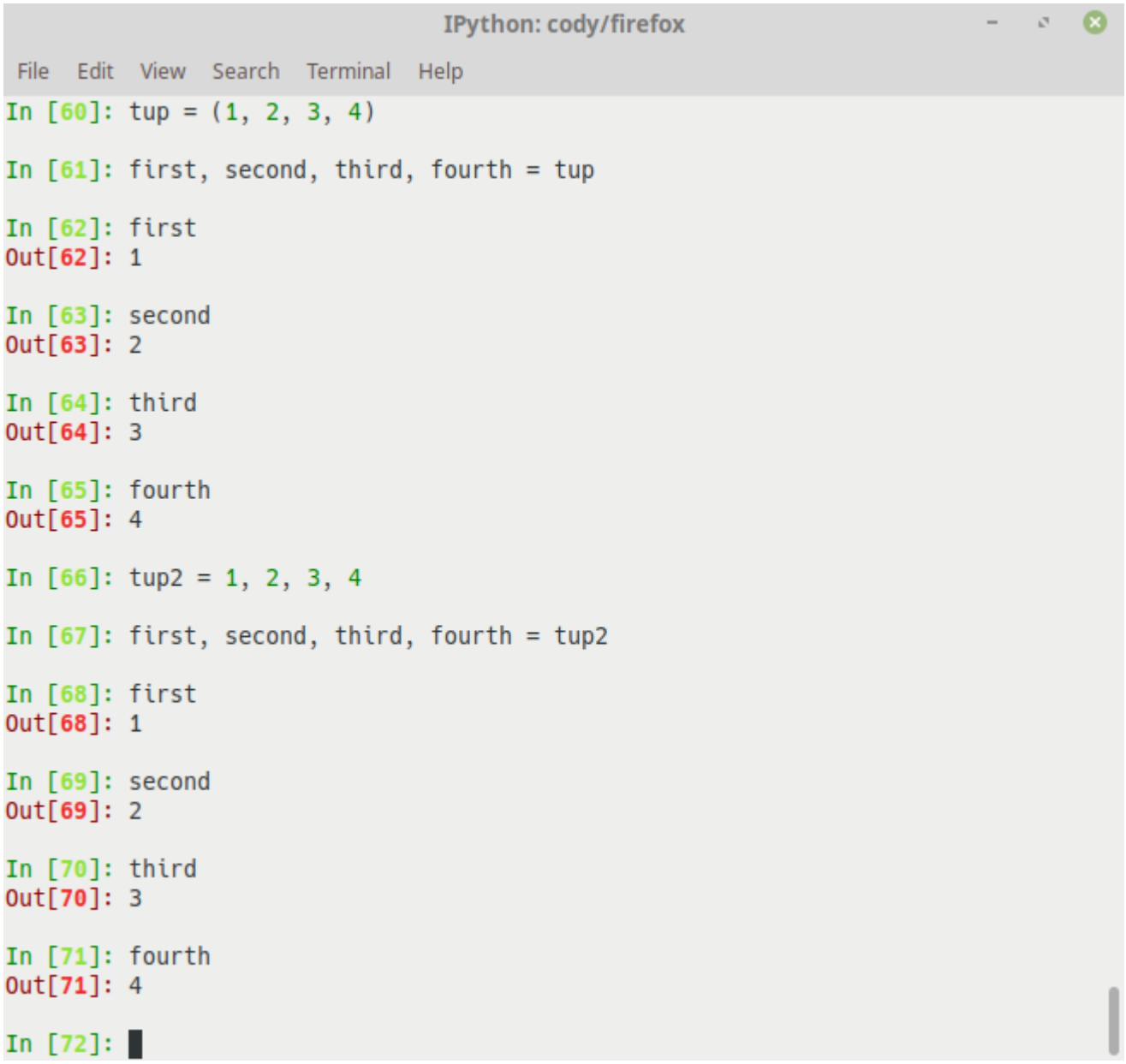


Tuples as keys

- Tuples are great when you want to return multiple values from a function. Normally, you can only return a single value from a function. If you return a tuple, however, multiple items can be placed into a single tuple object, so you aren't violating the single value rule, because it is a single tuple, yet you still get all the items that are contained in the tuple.

Sequence unpacking

To create a tuple, we simply create a variable and assign items to it, separated by commas. The term for this is packing a tuple, because the data is packed into the tuple, all wrapped up and ready to go. To remove items from a tuple, you simply unpack it, as shown in the following screenshot:



Unpacking a tuple

In line 60, the tuple is packed with a sequence of numbers, and in line 61, the items in the tuple (the numbers) are unpacked and assigned to individual variables. Lines 62-65 demonstrate that each number has been assigned to separate variables.

Line 66 shows the same thing, except the tuple parentheses, have been dropped to show that they aren't necessary.

Tuple unpacking is nice when you have a lot of items to work with. Rather than having a separate variable for each item, you can pack them all into a tuple and work with that. When you need to, you can unpack the tuple and work with the individual items directly.

One benefit of tuple packing/unpacking is that you can swap items in-place. With other languages, you have to create the logic to swap variables; with tuples, the logic is inherent in the data type, as shown in the following screenshot:

In-place variable swapping

Tuple unpacking and in-place swapping is one of the neatest features of Python, in my opinion. Rather than creating the logic to pull each item from a collection and place it in its own variable, tuple unpacking allows you to do everything in one step. In-place swapping is also a shortcut; you don't need to create temporary variables to hold the values as you switch places.

Sets

Sets are unordered collections of hashable objects; in other words, each object is unique. Sets are commonly used to see if a collection of objects contains a particular item, remove duplicates from a sequence, and compute a variety of mathematical operations.

Sets look like dictionaries, in that curly braces `{}` are used to create a set. However, unlike dictionaries, sets only have values; there are no key names within a set.

The following example shows how to create a set:

```
knights_set = {"Sir Galahad", "Sir Lancelot", "Sir Robin"}
```

Sets are also like dictionaries in that the objects they contain are unordered, and it is likely that calling a set will show a different order of objects compared to what was originally set.

There are actually two types of sets: **set** and **frozenset**. A regular set is mutable, in that it can be modified in-place. A frozenset is immutable and cannot be altered after creation. Therefore, a frozenset can be used as a dictionary key, like a tuple, but a regular set cannot.

Set-specific operations are covered in *Set methods*, though they can utilize many of the sequence methods listed in *Sequence methods*.

Using data type methods

Because everything in Python is an object, and the vast majority of objects in Python have methods to provide functionality, this section will discuss some of the more common methods available to Python data types.

Sequence methods

The following methods are common to most sequence types, such as lists, tuples, sets, and strings, except where indicated:

- `x in seq`: `True` if an item within the sequence is equal to `x`; otherwise, `False` is returned. This also applies to a subset of a sequence, such as looking for a specific character within a string.
- `x not in seq`: `True` if no item within the sequence is equal to *x*; otherwise, `False` is returned.
- `seq1 + seq2`: Concatenates two sequences; if immutable sequences, a new object is created.
- `seq * n`: Adds a sequence to itself *n* times.
- `seq[i]`: Returns the i^{th} item of a sequence, with the first object's index value = `0`.
- `seq[i:j]`: Returns a slice of the sequence, from `i` (inclusive) to `j` (exclusive). Not available with sets.
- `seq[i:j:k]`: Returns a slice of the sequence, from `i` (inclusive) to `j` (exclusive), skipping every `k` values. Not available with sets.
- `len(seq)`: Returns the length of a sequence; that is, the number of items within the sequence.
- `min(seq)`: Returns the smallest item in a sequence.
- `max(seq)`: Returns the largest item in a sequence.
- `seq.index(x[, i[, j]])`: Returns the index value for the first occurrence of value `x` in a sequence; optionally, the first occurrence at or after index value `i` but before index `j`. Not available with sets.
- `seq.count(x)`: Returns the total number of occurrences of `x` in a sequence. Not available with sets.

The following methods are common to all mutable sequence types, such as lists and strings, except where indicated:

- `seq[i] = x`: Item `i` within a sequence is replaced with `x`.
- `seq[i:j] = iter`: A slice of `seq`, from `i` (inclusive) to `j` (exclusive), is replaced with the contents of iterable object `iter`. Not available with sets.
- `del seq[i:j]`: Deletes the given slice in `seq`. Not available with sets.
- `seq[i:j:k] = iter`: A slice of the sequence, from `i` (inclusive) to `j` (exclusive), skipping every `k`values, is replaced by the contents of `iter`. Not available with sets.
- `del seq[i:j:k]`: Deletes a slice of `seq`, skipping every `k` value. Not available with sets.
- `seq.append(x)`: Appends `x` to the end of `seq`. Not available with sets; use `set.add(x)` instead.
- `seq.clear()`: Deletes all contents of `seq`.
- `seq.copy()`: Makes a new copy of `seq`.
- `seq.extend(iter)`: Extends the sequence with the contents of `iter`. Not available with sets; use `set.union(*others)` instead.
- `seq *= n`: Updates the sequence with *n* copies of itself. Not available with sets.
- `seq.insert(i, x)`: Inserts item `x` into the sequence at index value `i`.
- `seq.pop([i])`: Returns the item at index `i` and removes it from the sequence.
- `s.remove(x)`: Deletes the first item from `seq` that equals `x`.
- `s.reverse()`: Reverses the sequence in-place.

String methods

As there are more than 40 methods for strings, we will cover some of the most commonly found methods here. However, the full list can be found in the Python documentation:

- `str.capitalize()`: Returns a copy of the string with only the first character capitalized and all others lowercase.
- `str.endswith(suffix[, start[, end]])`: Returns `True` if the string ends with the specified `suffix`; otherwise, returns `False`. To look for multiple suffixes, a tuple can be used. The optional `start` is the index to start the search at, and the optional `end` is the ending index.
- `str.format(*args, **kwargs)`: Conducts a string formatting operation. This has been shown previously in other examples. There are many additional parameters that can be used with string formatting, so the official documentation should be referenced. On an additional note,
  the `*args` and `**kwargs` arguments are frequently found in the Python documentation. They simply indicate what types of arguments are accepted by a function or method. For `*args`, any argument passed in will be processed; `**kwargs` indicates key=value arguments are accepted. Naturally, if the argument passed in is not known, an error will be generated.
- `str.isalpha()`: Returns `True` if all characters in the string are alphabetic. There are also methods for alphanumeric, numbers, ASCII-only, lowercase, and so on, checks.
- `str.ljust(width[, fillchar])`: Returns a string that is left-justified with a length of `width`. By default, any extra space is padded with space characters, but `fillchar` can be used to provide alternative characters.
- `str.lower()`: Returns a copy of the string that is all lowercase.
- `str.splitlines([keepends])`: Returns a list of the individual lines within a string, as determined by common line separator characters. The line breaks themselves are not included, unless `keepends` is `True`.
- `str.strip([chars])`: Returns a copy of the string with the lead and trailing characters removed. By default, all whitespace is removed, but the optional `chars` argument can specify specific characters to remove.
- `str.title()`: Returns a copy of the string in title case. Due to the algorithm used, apostrophe characters can cause problems with the expected output, so a review of the documentation is suggested prior to use.

As previously mentioned, these are not all the special methods available to strings. Also, because strings are simply specialized lists, they also have access to all the sequence methods listed in *Sequence methods*.

List methods

Lists have only one special method: `list.sort(*, key=None, reverse=False)`. By default, `sort()` performs an *A*-*Z*-style sort, with lower values on the left. Using `key` allows the use of an additional function to modify the default sort, while `reverse` performs a *Z*-*A* sort. An example of these sorting operations is shown in the following screenshot:

List sorting methods

In line 145, a list of strings is created and the default sort is performed in line 146. The results of the default sort are shown in line 147. Note that capital letters come before lowercase ones.

In line 148, the `key` modifier is used to take lowercase letters into account, resulting in the unusual results of line 149. These are unusual because, rather than simply moving the lowercase words to the beginning of the list, the `sort()` method has actually sorted the list according to the actual alphabetical listing of the beginning characters, regardless of their capitalization.

Line 150 performs a reverse sort. The output in line 151 shows that capitalization doesn't matter, as we didn't specify a `key`, so the results are simply the opposite of line 147.

Tuple methods

Tuples implement all the common sequence methods (see *Sequence methods*), except for the mutable-only ones.

Dictionary methods

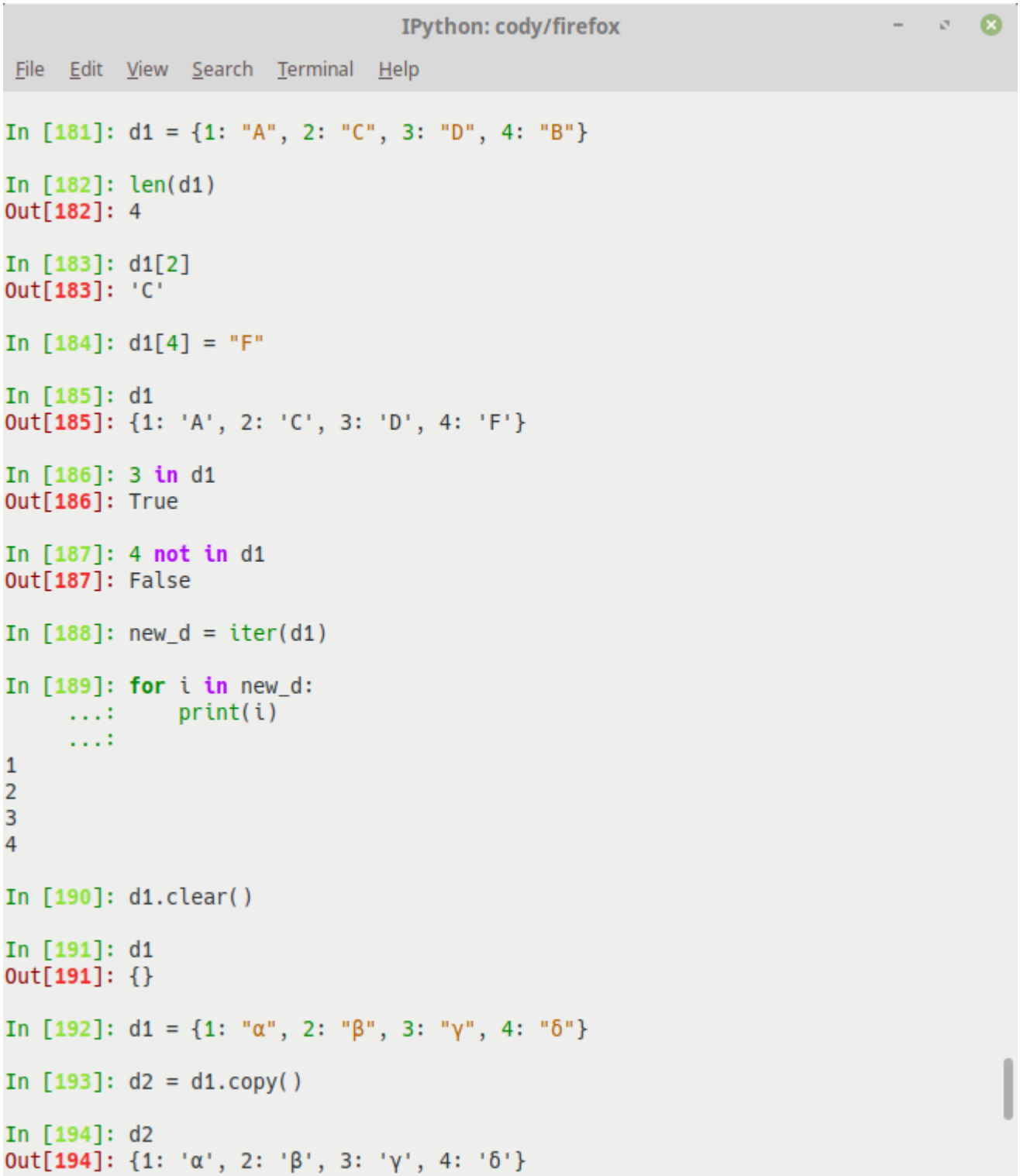
Dictionaries, as hashable arrays rather than sequences, have a number of methods that are unique to them, and only a few of the sequence-specific methods are similar. The following is a list of the dictionary-specific methods:

- `len(dict)`: Returns the number of items in a dictionary.
- `dict[key]`: Returns the value associated with `key`. If the specified `key` is not in the dictionary, an error is generated.
- `dict[key] = value`: Sets the association of `key` to `value`.
- `del dict[key]`: Deletes the indicated `key`, and its associated value, from the dictionary. If `key`doesn't exist, an error is generated.
- `key in dict`: If the dictionary has the specified key name, returns `True`; otherwise, returns `False`.
- `key not in dict`: If the dictionary does not contain the specified key name, returns `True`; otherwise, returns `False`.
- `iter(dict)`: Returns an iterator object that contains the keys of the dictionary.
- `dict.clear()`: Deletes all entries from the dictionary.
- `dict.copy()`: Returns a copy of the dictionary.
- `dict.fromkeys(seq[, value])`: Creates a new dictionary, with the key names taken from a sequence. The optional `value` defines the default value associated with each key.
- `dict.get(key[, default])`: Returns the value for `key` if the key exists; if the key doesn't exist, the optional `default` value is returned. If `default` is not defined, then `None` is returned; this prevents an error from being generated.
- `dict.items()`: Returns a dictionary view object of all the key:value pairs from the dictionary.
- `dict.keys()`: Returns a dictionary view object of just the key names from the dictionary.
- `dict.pop(key[, default])`: Returns the value for `key` if the key exists, while removing the key:value pair from the dictionary. If the key doesn't exist, then the optional `default` value is returned to prevent an error.
- `dict.popitem()`: Returns a key:value pair from the dictionary while removing it from the dictionary. Pairs are removed in a last-in, first-out order.
- `dict.setdefault(key[, default])`: Returns the value associated with `key` if it exists in the dictionary. If it doesn't exist, `key` is added to the dictionary and its value is set to the optional `default`value, then return the `default` value. If no `default` value is provided, it defaults to `None`.
- `dict.update([other])`: Updates the dictionary with the key:value pairs, as defined in `other`, by overwriting existing items. `update` accepts either a dictionary or an iterable object of key:value pairs, such as a tuple.
- `dict.values()`: Returns a dictionary view object of all the values existing in the dictionary.

When an item states it provides a *dictionary view object*, it just means that it provides a view of the indicated part of a dictionary, and this view is updated in real time. Views are not copies of the objects, but simply a display of the current contents.

For example, `dict.keys()` shows all the current keys associated with a particular dictionary; if the keys are updated, `dict.keys()` will show the change.

The aforementioned methods are demonstrated in the following examples, which are split into two screenshots due to the number of entries:


IPython: cody/firefox
File Edit View Search Terminal Help
In [181]: d1 = {1: "A", 2: "C", 3: "D", 4: "B"}
In [182]: len(d1)
Out[182]: 4
In [183]: d1[2]
Out[183]: 'C'
In [184]: d1[4] = "F"
In [185]: d1
Out[185]: {1: 'A', 2: 'C', 3: 'D', 4: 'F'}
In [186]: 3 in d1
Out[186]: True
In [187]: 4 not in d1
Out[187]: False
In [188]: new_d = iter(d1)
In [189]: for i in new_d:
...: print(i)
...:
1
2
3
4
In [190]: d1.clear()
In [191]: d1
Out[191]: {}
In [192]: d1 = {1: "α", 2: "β", 3: "γ", 4: "δ"}
In [193]: d2 = d1.copy()
In [194]: d2
Out[194]: {1: 'α', 2: 'β', 3: 'γ', 4: 'δ'}
Dictionary methods, part 1

Line 181 creates our dictionary. Line 182 displays the number of key:value pairs within the dictionary.

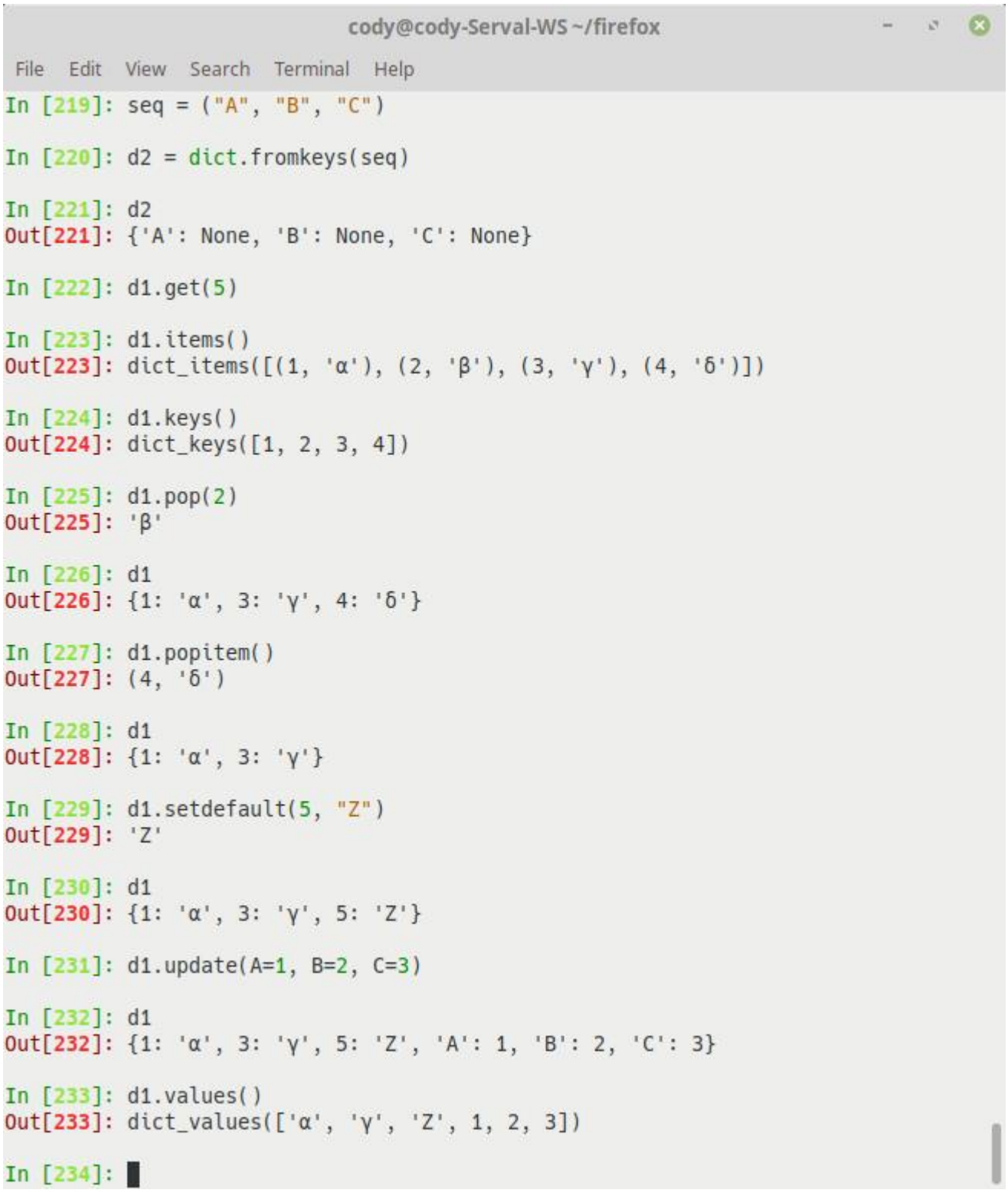
Line 183 returns the value for the key named `2`, while line 184 changes the value associated with key `4`, as shown in line 185.

To check whether a particular key exists in the dictionary, lines 186 and 187 provide alternative methods to do that.

Line 188 iterates over the dictionary and assigns the keys to a new sequence. Line 189 iterates over the sequence and prints the keys that were pulled out. Note that the sequence `new_d` is not a dictionary, but is actually a different type: `dict_keyiterator`. An iterator is a special object that is associated with iterations and isn't normally defined by a programmer, except in certain circumstances that won't be covered here.

Line 190 removes all entries from the dictionary, as shown in line 191, but we add new entries in line 192. Note that Python is comfortable with non-ASCII characters, such as Greek letters. Python 3 deals with Unicode characters by default, so any characters can be used, not just the normal English alphabet.

In line 193, a new dictionary is made as a copy of `d1`, as shown in line 194:


cody@cody-Serval-WS ~/firefox
File Edit View Search Terminal Help
In [219]: seq = ("A", "B", "C")
In [220]: d2 = dict.fromkeys(seq)
In [221]: d2
Out[221]: {'A': None, 'B': None, 'C': None}
In [222]: d1.get(5)
In [223]: d1.items()
Out[223]: dict_items([(1, 'α'), (2, 'β'), (3, 'γ'), (4, 'δ')])
In [224]: d1.keys()
Out[224]: dict_keys([1, 2, 3, 4])
In [225]: d1.pop(2)
Out[225]: 'β'
In [226]: d1
Out[226]: {1: 'α', 3: 'γ', 4: 'δ'}
In [227]: d1.popitem()
Out[227]: (4, 'δ')
In [228]: d1
Out[228]: {1: 'α', 3: 'γ'}
In [229]: d1.setdefault(5, "Z")
Out[229]: 'Z'
In [230]: d1
Out[230]: {1: 'α', 3: 'γ', 5: 'Z'}
In [231]: d1.update(A=1, B=2, C=3)
In [232]: d1
Out[232]: {1: 'α', 3: 'γ', 5: 'Z', 'A': 1, 'B': 2, 'C': 3}
In [233]: d1.values()
Out[233]: dict_values(['α', 'γ', 'Z', 1, 2, 3])
In [234]:
Dictionary methods (continued)

Line 219 defines a tuple of strings. This tuple is used in line 220 to overwrite the items in dictionary `d2`. As shown in line 221, the strings from the tuple are now the keys in the dictionary, and the values have defaulted to `None`.

Line 222 attempts to retrieve the value associated with the key `5` but, since there is no key `5` in dictionary `d1`, nothing is returned. While this may not seem special, using the `get()` method rather than

the normal `d1[5]` command, as shown in line 183, means that no error is generated if the specified key doesn't exist.

Line 223 shows all the items that are in the dictionary, while line 224 shows just the keys.

Line 225 pulls the requested key from the dictionary and returns its value. The key:value pair is deleted from the dictionary, as shown in line 226. In line 227, the last key:value pair in the dictionary is removed and returned, no longer existing within the dictionary, as shown in line 228.

Line 229 requests the value associated with key `5`. Since that key doesn't exist, the specified value of `Z` is entered into the dictionary and summarily returned as the value associated with `5`. The addition of the new key:value pair is shown in line 230.

In line 231, the dictionary is updated with new key:value pairs; had any of the keys been a duplicate of an existing key, that key:value pair would have been overwritten. As such, the new additions are placed into the dictionary, as shown in line 232. Note that the sequence used in line 231 does not use quotation marks for the strings. In this case, Python is smart enough to understand what is desired and will perform the operation without errors. As a matter of fact, attempting to use quotation marks may generate an error.

Finally, line 233 shows all the values that exist within the dictionary.

Dictionaries are very powerful and useful data types, and are used for many different things. Learning to use them well is an important skill for anyone who wants to code well.

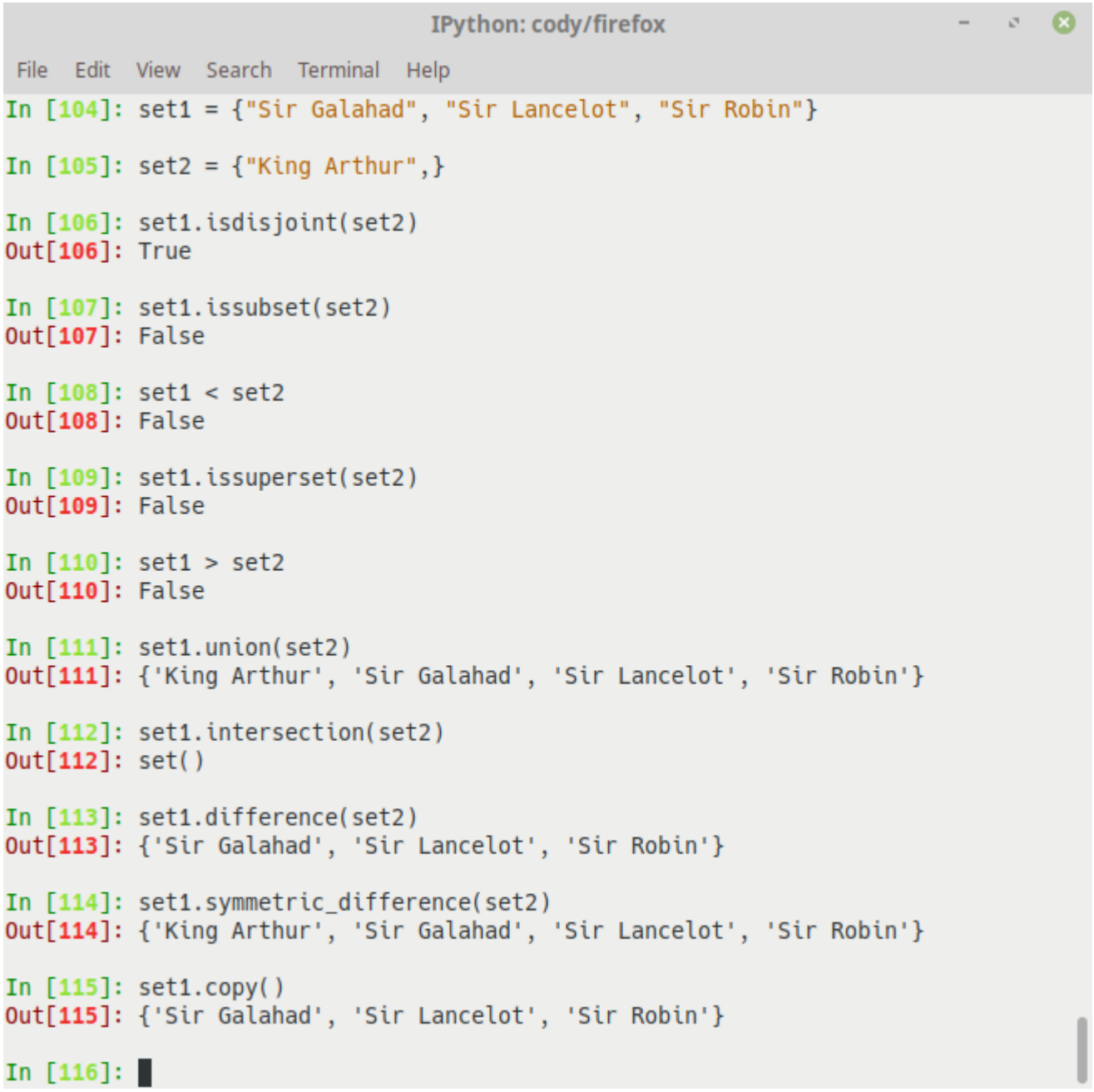
Set methods

Operations that are specific to sets and frozensets generally provide a way to quickly compare and shift out common/uncommon items between the different sets. Examples of the following methods can be seen in the next screenshot. The following non-exhaustive listing of set methods is an example of the more common set methods.

However, be sure to review the official Python documentation as there are some differences between set and frozenset methods:

- `set1.isdisjoint(set2)`: Returns `True` if `set1` has no elements in common with `set2`
- `set1.issubset(set2)`: Returns `True` if every element in `set1` exists in `set2`
- `set1 < set2`: Returns `True` if `set1` is a true subset of `set2` but not exactly equal to `set2`
- `set1.issuperset(set2)`: Returns `True` if every element in `set2` is in `set1`
- `set1 > set2`: Returns `True` if `set1` is a true superset of `set2` but not exactly equal to `set2`
- `set1.union(set2, set3, ...)`: Returns a new set that includes elements from all given sets
- `set1.intersection(set2, set3, ...)`: Returns a new set with all common elements between the given sets
- `set1.difference(set2, set3, ...)`: Returns a new set with elements that exists in `set1` but are not in any others
- `set1.symmetric_difference(set2)`: Returns a new set with elements that are unique to each set
- `set1.copy()`: Returns a new set with a copy of the elements from `set1`

The following screenshot shows an example of set method:



Set method examples

Lines 104 and 105 create two different sets. Lines 106-110 are self-explanatory, based on the previous definitions.

With line 111, we create a new set by merging `set1` with `set2`. Line 112 shows a returned, empty set because there are no common elements between the two sets.

Line 113 shows the elements that exist in `set1` but not `set2`, while line 114 shows all the unique elements.

Finally, line 115 presents a copy of `set1`; normally, this would be assigned to a new variable for later use.

Importing modules

We briefly touched on importing modules way back in `Chapter 1`, *The Fundamentals of Python*. Modules are also called libraries or packages. Modules are modular, often self-contained Python programs that are commonly utilized in other programs, hence the need to import them for access.

Modules are used to separate code to make a program easier to work with, as each module can be designed to do one thing well, rather than having to make a single program that is responsible for all logic.

The **Python Package Index** (**PyPI**) website (`https://pypi.org`) is the official repository of third-party Python libraries. At the time of writing, there are more than 150,000 packages available for download from PyPI. Most of these packages are designed to be imported into a Python project to provide additional, or easier, functionality than can be achieved with the default Python libraries.

Namespaces

Another benefit of modules is that they create additional namespaces for code. Namespaces (also called *scopes*) are the hierarchy of control that a module has. Normally, objects outside of a module aren't visible to code within the module; thus, they can't be accessed or utilized within the current module.

The benefit of this segregation is that variable shadowing is less likely. Variable shadowing is the creation of duplicate variable names in different blocks of code, such that one variable is hidden (shadowed) by an identical variable and cannot be accessed, or the Python interpreter may call the incorrect variable.

Using a module allows the same variable name to be used in multiple locations without requiring shadowing to occur, as a specific variable is identified by the module it resides in. Of course, there is nothing to stop a programmer from using the same variable within a module, with the potential of shadowing occurring, but the namespace hierarchy makes that unlikely.

Global variables are an option, but aren't recommended. Global variables allow a programmer to define a variable that can be accessed within any namespace; they are commonly used to contain data that is used in multiple locations, such as a counter. Of course, this leaves open the possibility that a global variable will be overwritten without the programmer realizing, causing a problem later on in the program.

Program scope works inside-out. As a module is typically made with multiple functions or methods, when an object is called, the Python interpreter will look for the correct reference within the current function/method. If the object isn't defined there, the interpreter will move to the enclosing container, if one exists (such as another function or a method's class). If the variable can't be found there, the interpreter looks for a global variable. Not finding one, the interpreter will look in the built-in libraries. If still not found, Python will generate an error. The flow looks like this: local container|enclosing container|global scope|built-in module|error.

The following is a simple program that should help explain this idea a little better. We haven't directly talked about functions or `if...else` statements yet, but hopefully this won't be too confusing:

```
# scope_example.py (part 1)
1  var1 = 1  # global variable
2
3  if var1 == 1:
4     var2 = 0  # also a global variable
5     print("Unmodified var2: {}".format(var2))
6
7  def my_funct():
8    var3 = 3  # local variable
9    var1 = 42  # shadows global var1 variable
10   global var2
11   var2 = 80
12   print("Inside function, var1: {}".format(var1))
13   print("Inside function, var2: {}".format(var2))
14   print("Inside function, var3: {}".format(var3))
```

In line 1, a global variable is created. Line 3 is a test to see whether `var1` is equal to `1`; if so, then a new global variable (`var2`) is created, and its value is printed.

Line 7 is the start of a Python function. Within this function, a variable that is only accessible within the function is created (`var3`). In addition, a new variable (`var1`) is created; this variable hides the previously made global variable `var1`, so when the value of this local `var1` is printed in line 12, the local value is printed, rather than the value of the global variable.

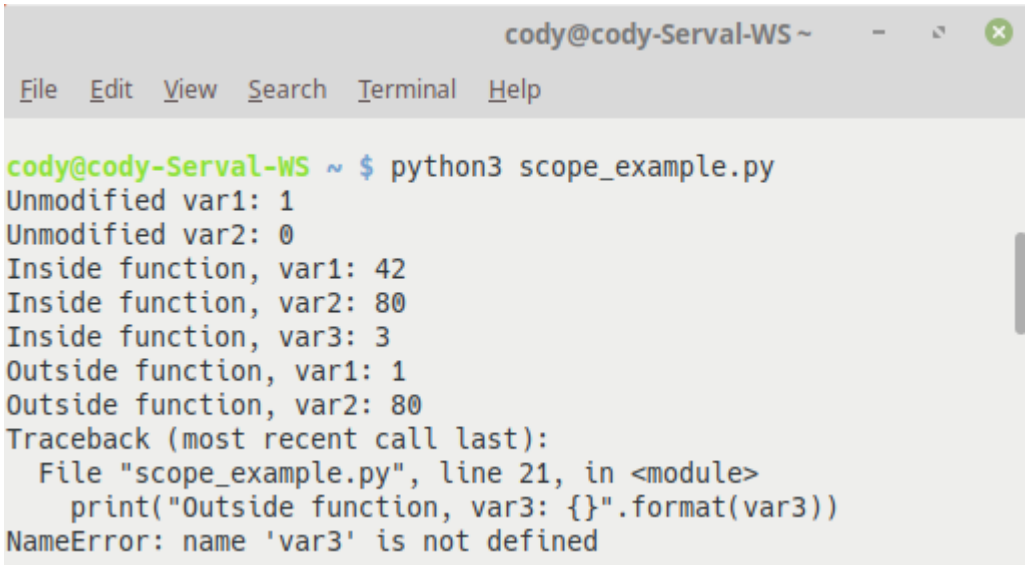
Line 10 explicitly calls the global variable `var2`; this allows the function to manipulate the global variable in line 11 without attempting to make a local variable that would shadow it.

Lines 12-14 print the values of the variables as seen within the scope of the function:

```
# scope_example.py (part 2)
1 my_funct()
2
3 print("Outside function, var1: {}".format(var1))
4 print("Outside function, var2: {}".format(var2))
5 print("Outside function, var3: {}".format(var3))
```

In the second part, line 1 is the call to the function to actually run it. *Lines 3-5* print the values of the variables as seen outside the function.

The following screenshot displays the output of the previous program.


cody@cody-Serval-WS ~
File Edit View Search Terminal Help
cody@cody-Serval-WS ~ $ python3 scope_example.py
Unmodified var1: 1
Unmodified var2: 0
Inside function, var1: 42
Inside function, var2: 80
Inside function, var3: 3
Outside function, var1: 1
Outside function, var2: 80
Traceback (most recent call last):
File "scope_example.py", line 21, in <module>
print("Outside function, var3: {}".format(var3))
NameError: name 'var3' is not defined
Output of scope_example.py

The `print()` calls show the value of each variable at its respective location within the program. Initially, `var1` and `var2` are the values of the globally defined variables. Once the function has been called and performs its operations, the local `var1` and `var3` variables are printed, along with the global `var2` variable, whose value has been replaced.

When the function is complete and we are back outside the function, we see that the globally defined variable `var1` is back to its original value, and is no longer hidden by the local function variable of the same name. However, because `var2` was explicitly called to reference the global variable, rather than shadow it, `var2` retains the value it was assigned while within the function.

Finally, because `var3` doesn't exist outside the function, the interpreter doesn't know what to do with it. Since we are no longer within the function, there is no local reference to it. Moving up the namespace, there is no encapsulating function or other object, and there is no global reference to `var3`. Since Python doesn't have a `var3` object in any of its built-in libraries, the only thing Python can do with the call is to give up and throw an error.

Dot nomenclature

When a module is imported, after the local and global checks within the current program, the imported module will be examined for the called object as well (prior to checking the built-ins). The problem with this is that, unless the called object is explicitly identified with the dot nomenclature, an error will still be generated.

In the following screenshot, we see how the dot nomenclature works:

Dot nomenclature

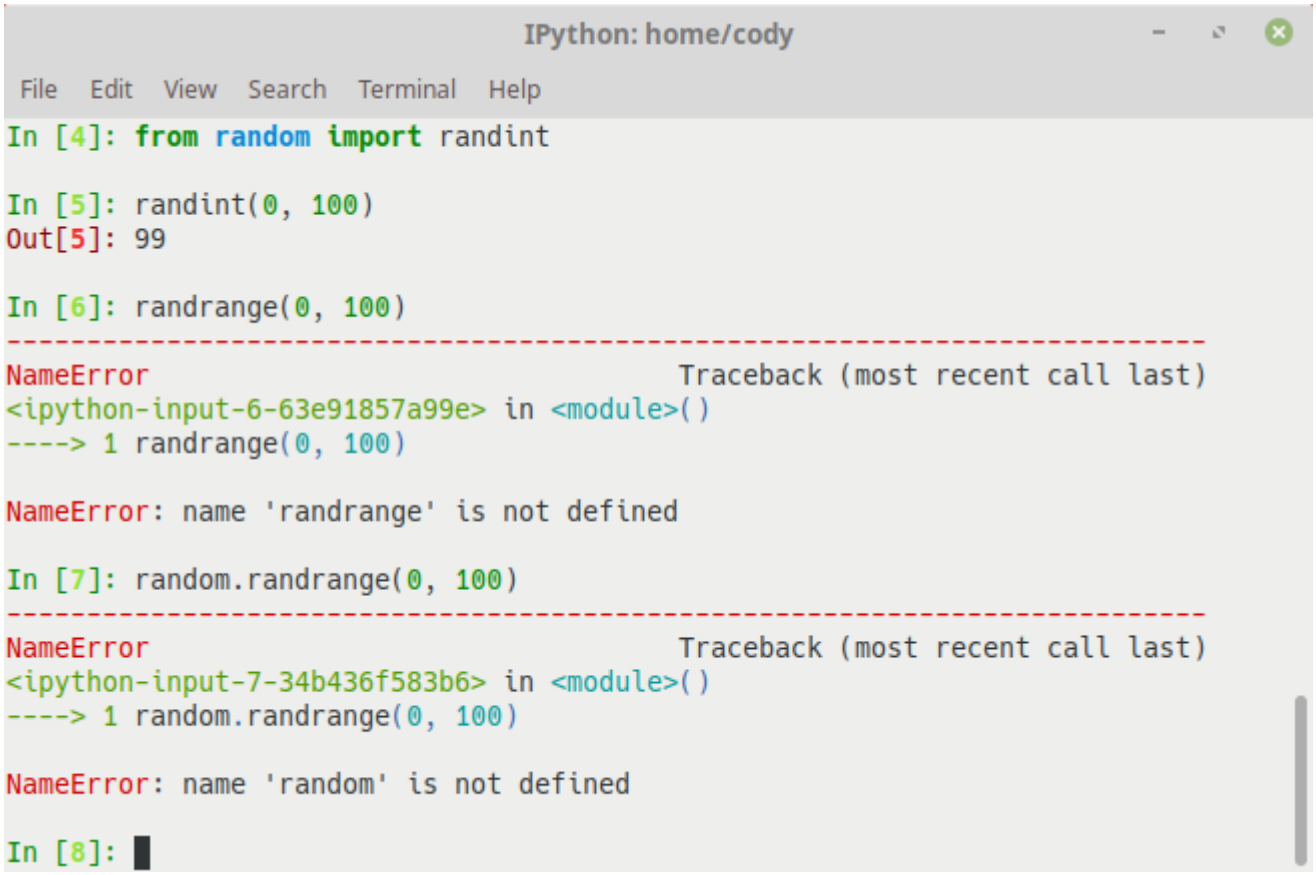
In this case, we attempt to calculate the square root of a number. Since this operation is unknown to the default Python interpreter, an error is generated. However, if we import the `math` library in line 2, and then attempt to perform the calculation again, we get an answer.

Note that we explicitly told Python that the square root function is to be found in the `math`library by using the `math.sqrt()` command. This is the dot nomenclature that we talked about earlier; the dot indicates that the `sqrt()` function can be found in the `math` library. We will see many other examples of this as we discuss programming further, so while it may not make sense right now, hopefully more examples will help.

Types of imports

In the previous screenshot, we performed a basic module import. This just means that we imported the module, and then referenced something within it through the dot nomenclature. With this type of import, the main program and the imported module maintain their separate namespaces, hence the need to explicitly identify a function through the dot nomenclature.

There are other ways to import modules. One of the most common is to use the *from* version of import to get only specified objects from a module, rather than the entire module, as shown in the following screenshot.



From import

In this case, we are importing just `randint()` from the `random` library in line 4. Because we have explicitly imported this function, it is now part of the overall program's namespace, rather than being separated into the random namespace and requiring the dot nomenclature to call it. This allows us to call it in line 5 without any special conditions.

If we try to do the same for `randrange()` in line 6, we get an error because we never imported `randrange()` explicitly. Even if we try to use the dot nomenclature in line 7, we still get an error because the entire `random` library was not imported.

One way around this is to use the `from <module> import *` command, which imports nearly all objects from the specified module. The problem with this is the possibility of name shadowing because of all the imported objects, especially if multiple modules are imported this way.

In general, if only a handful of objects are needed, explicitly importing them is the safest way to work with them. If you need most or all of a library, you can use the `import *` command (it's easier to work with but not as safe) or the dot nomenclature (which is safer but requires more typing).

Modules as scripts

An important thing to know about Python is that modules, as written, are pretty much only useful as imported objects for other programs. However, a module can be written to be imported or function as a standalone program.

When a module is imported into another program, only certain objects are imported over. Not everything is imported, which is what allows a module to perform dual duty. To make a module operate by itself, a special line has to be inserted near the end of the program.

The following program is a simple dice rolling simulator, broken up into separate parts:

```
# random_dice_roller.py (part 1)
1  import random #randint
2
3  def randomNumGen(choice):
4      """Get a random number to simulate a d6, d10, or d100 roll."""
5
6      if choice == 1: #d6 roll
7          die = random.randint(1, 6)
8      elif choice == 2: #d10 roll
9          die = random.randint(1, 10)
10     elif choice == 3: #d100 roll
11         die = random.randint(1, 100)
12     elif choice == 4: #d4 roll
13         die = random.randint(1, 4)
14     elif choice == 5: #d8 roll
15         die = random.randint(1, 8)
```

The preceding code imports the `random` library from the built-in Python modules. Next, we define the function that will actually perform the dice simulation in line 3:

```
# random_dice_roller.py (part 2)
1      elif choice == 6: #d12 roll
2         die = random.randint(1, 12)
3      elif choice == 7: #d20 roll
4         die = random.randint(1, 20)
5      else:   #simple error message
6          return "Shouldn't be here.  Invalid choice"
7      return die
8
9 def multiDie(dice_number, die_type):
10      """Add die rolls together, e.g. 2d6, 4d10, etc."""
11
```

```
12 #---Initialize variables
13     final_roll = 0
14     val = 0
```

In the preceding code, we continue the different dice rolls and then define another function (line 9) that combines multiple dice together, as frequently used in games:

```
# random_dice_roller.py (part 3)
1    while val < dice_number:
2        final_roll += randomNumGen(die_type)
3        val += 1
4    return final_roll
5
6 if __name__ == "__main__":  #run test() if calling as a separate program
7     """Test criteria to show script works."""
8
9     _1d6 = multiDie(1,1)   #1d6
10    print("1d6 = ", _1d6, end=' ')
11    _2d6 = multiDie(2,1)   #2d6
12    print("\n2d6 = ", _2d6, end=' ')
13    _3d6 = multiDie(3,1)   #3d6
14    print("\n3d6 = ", _3d6, end=' ')
```

In the preceding code, we finish with the summation of dice. The key part of the entire program is line 6. This line determines whether the module can run by itself or can only be imported into other programs.

Line 6 states that, if the namespace seen by the interpreter is the main one (that is, if `random_dice_roller.py` is the main program being run and not something else), then the interpreter will process any operations that are specified below line 6. In this case, these operations are simply tests to confirm that the main logic (preceding line 6) works as expected.

If this program were to be imported into another program, then everything before line 6 would be imported while everything following it would be ignored. Thus, you can make a program that functions as a standalone program or can be imported; the only difference is what code is written below `if __name__ == "__main__"`:

```
# random_dice_roller.py (part 4)
1    _4d6 = multiDie(4,1)   #4d6
2    print("\n4d6 = ", _4d6, end=' ')
3    _1d10 = multiDie(1,2)   #1d10
4    print("\n1d10 = ", _1d10, end=' ')
5    _2d10 = multiDie(2,2)   #2d10
6    print("\n2d10 = ", _2d10, end=' ')
7    _3d10 = multiDie(2,2)   #3d10
8    print("\n3d10 = ", _3d10, end=' ')
9    _d100 = multiDie(1,3)   #d100
10    print("\n1d100 = ", _d100)
```

This finishes the self-tests for the dice rolling program.

Summary

In this chapter, we discussed how to structure Python code, including how to span multiple lines, if necessary. Next, we covered the various data types that are included in Python: numbers, strings, lists, dictionaries, tuples, and sets. We also demonstrated how to use those data types to make simple scripts, and then talked about frequently used methods that provide more functionality to the data types. Finally, we saw how to import modules and how they affect the ability to interact with different parts of Python code.

In the next chapter, we will learn how to control logic flow within a program using if...else statements, looping, and dealing with error exceptions.

Logic Control

The main part of programming is learning how to make your code do something, primarily through a variety of logic controllers. These controllers handle `if-then` conditions, reiterative processing through loops, and dealing with errors. While there are other ways of working with code, these are the most important ones for new programmers to learn.

When dealing with logic control, a developer needs to be aware of how data is being transferred, particularly when working with user input, network connections, or filesystem access. Python has three data streams for **input/output** (**I/O**). `sys.stdout` is the standard output stream; it handles the output of `print()` and Python expressions. `sys.stdin` is the standard input stream; it is used for all interactive input. `sys.stderr` is the standard error stream; it only takes errors from the program, but also handles the interpreter's own prompts.

One thing to recognize is that, depending on the OS environment, information that you would consider going to `stdout` is actually sent to `stderr`, because `stderr` normally goes to the same location as `stdout`, by default you'll usually see it on the screen as well. However, you won't know that a response is actually going to `stderr` without testing. If the particular environment routes `stderr` to another location, such as a log file, you won't know until you need to troubleshoot. This is important to note because, sometimes, you may not be seeing the information you expect because it's not a normal `stdout` message.

These data streams are considered regular text files and can be accessed and interacted with just like normal files. File operations are looked at in `Chapter 5`, *Files and Databases*, in the *File I/O* section.

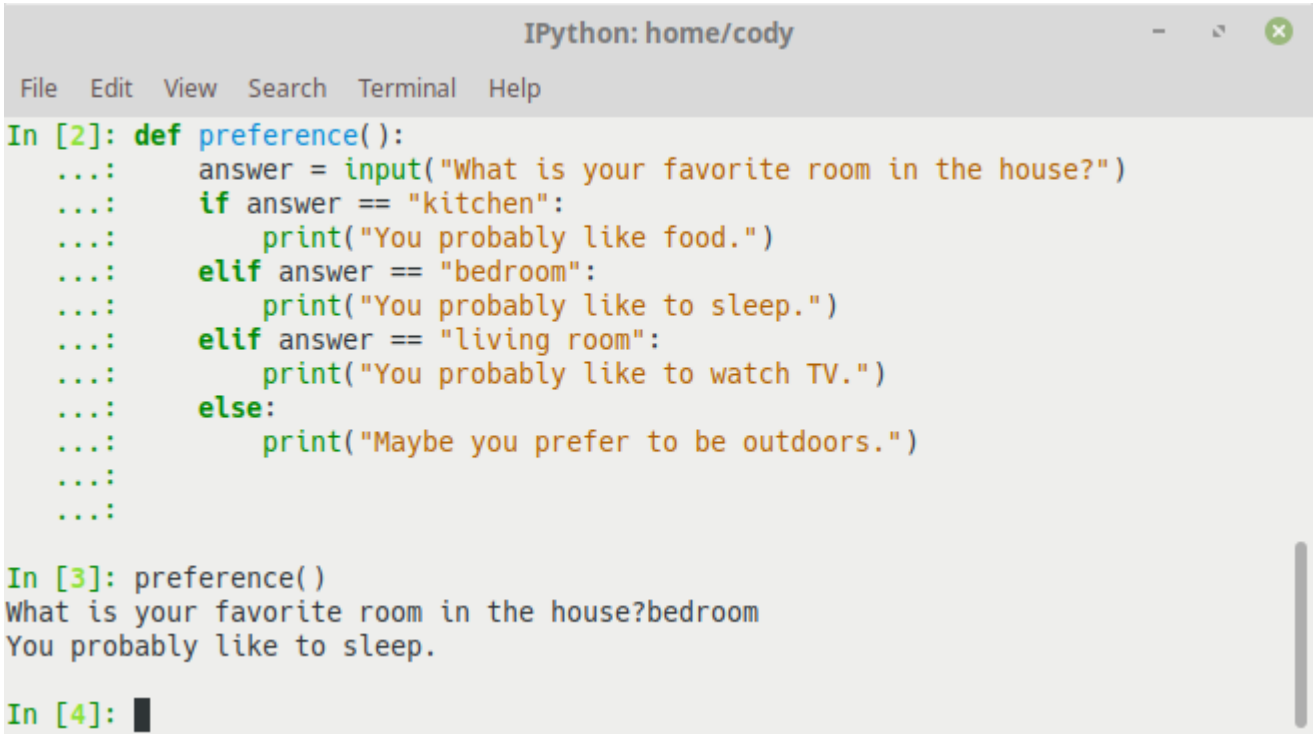
This chapter will cover the following:

- `if...else` statements
- Loops
- Exceptions

if...else statements

One of the most common control structures you'll use, and run into in other programs, is the `if...else` conditional block. Simply put, the program asks a yes or no question; depending on the answer, different things happen.

If you've programmed in other languages, the `if...else` statement works the same way. The key difference is that, in Python, the `elseif` statement is written as `elif` for checking multiple conditions, as shown in following screenshot:


IPython: home/cody
File Edit View Search Terminal Help
In [2]: def preference():
...: answer = input("What is your favorite room in the house?")
...: if answer == "kitchen":
...: print("You probably like food.")
...: elif answer == "bedroom":
...: print("You probably like to sleep.")
...: elif answer == "living room":
...: print("You probably like to watch TV.")
...: else:
...: print("Maybe you prefer to be outdoors.")
...:
...:
In [3]: preference()
What is your favorite room in the house?bedroom
You probably like to sleep.
In [4]:
Using if...else statements

In the preceding example, the `preference()` function is used to hold the main code logic; functions are explained in `Chapter 4`, *Functions and Object Oriented Programming*, in the *Working with functions* section. The `input()` function prints the string within parentheses to the user (normally a question), and accepts the user's input, and that input is assigned to the `answer` variable.

When checking for a `yes` or `no` condition, the only required part is the `if` statement. The `elif`(else/if) and the `else` statement aren't necessary. Having the `else` statement as a catch-all, default case is useful, especially if used with a `print()` command to indicate when an unexpected condition is received.

An `if` statement can be standalone, as shown here:

```
x = True
y = False
if x == True:
    y = True
    print(y)
```

More common is an `if...else` block, to have two different options depending on the condition, as shown here:

```
x = True
y = False
if x == True:
    y = True
    print(y)
else:
    print("'x' is not True")
```

Python doesn't have a `switch` or `case` statement, unlike other languages. A `switch` statement is a type of control device that allows a single variable to determine the rest of the program execution based on the

variable's value. An example of the `switch` statement from the C language is shown in the following example:

```
switch (grade) {
  case "A":
                printf("Outstanding!");
        break;
  case "B":
                printf("Good job!");
        break;
  case "C":
                printf("Satisfactory performance.");
        break;
  case "D":
                printf("You should try harder.");
        break;
  case "F":
                printf("You failed.");
                break;
  default:
                printf("Invalid grade");
}
```

While this is somewhat simplistic, you can probably see that a more complicated example could provide different branches to the rest of the program, if desired. The key point is that a single variable is tested, and the results of that test are compared to a variety of options; the option that matches dictates how the program continues.

You can get the same functionality of `switch` statements by using `if...elif` tests, searching within lists, or indexing dictionaries. Since lists and dictionaries are built at runtime, they can be more flexible. Following screenshot demonstrates how a dictionary can be used to perform the same functionality as a `switch` statement:

Dictionary as a switch statement

Obviously, this isn't the most intuitive way to write this program. A better way would be to create the dictionary as a separate object, and then use a dictionary method such as `key` in `dict`to find the value corresponding to your choice. In this case, you could use `"spam"` in choice.

However, it's more common to use `if...else` statements to perform this operation, as it looks similar to the normal `switch` choices and is the easiest way to deal with choices. The benefit to using a dictionary is that dynamic programs can create these data structures relatively easily. With `if...else` statements, they have to be written by the programmer prior to running the program, whereas dictionaries can be populated and tested programmatically during runtime.

Loops

There are a number of differing looping constructs available in Python, though `for` and `while`loops are the dominant ones. Generally speaking, loops allow a program to perform the same operation multiple times until a condition occurs that cancels the loop. Along with `if...else`statements, loops handle a large portion of program logic.

while loops

The `while` loop is a standard workhorse of many languages. Essentially, the program will continue doing something while a certain condition exists. As soon as that condition is no longer true, the loop stops.

`break` and `continue` work exactly the same as in C. The equivalent of C's empty statement (a semicolon) is the `pass` statement, and Python includes an `else` statement for use with breaks.

The `break` statements simply force the loop to quit early; when used with nested loops, it only exits the innermost enclosing loop. The `continue` statements cause the loop to start over with the next iteration, regardless of any other statements further on in the loop. The `else`statement block is run "on the way out" of the loop, unless a `break` statement causes the loop to quit early.

The following example demonstrates a generic `while` loop:

```
1 while <test>:
2     <statements>
3      if <test>:
4          break
5      elif <test>:
6          continue
7      else:
8          <statements>
```

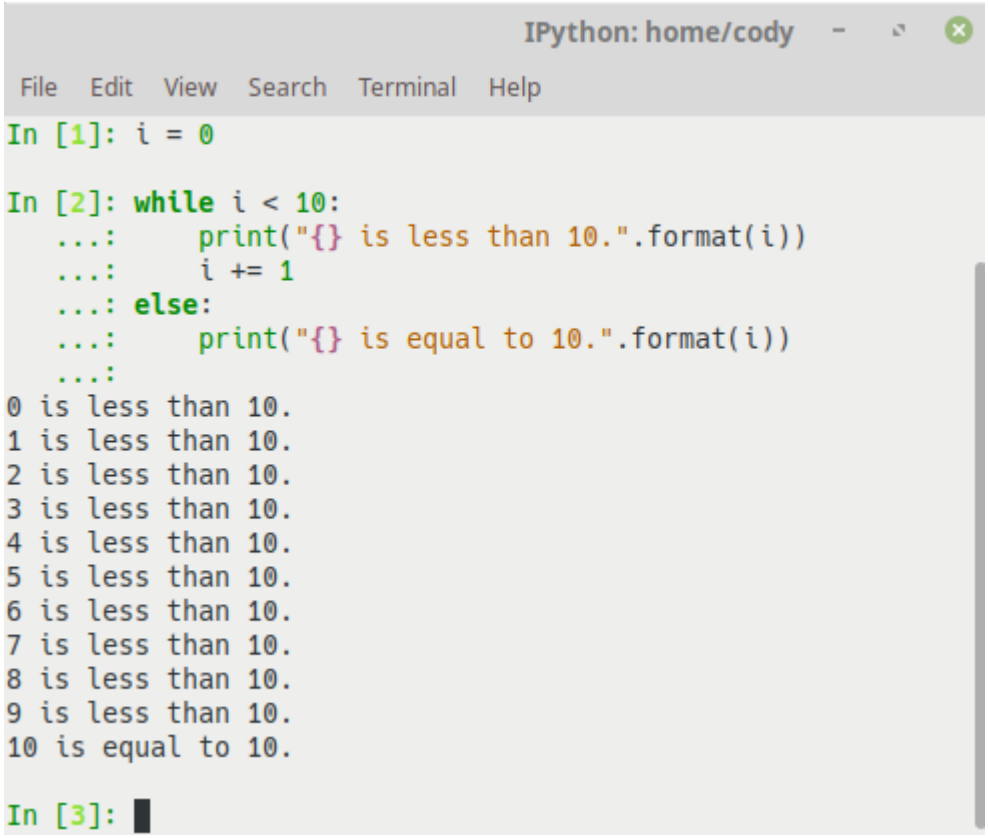
Line 1 declares the condition that is tested. As long as this condition is true, the loop will continue. Once it is no longer true, the loop quits.

Line 2 indicates the main logic of the loop that is performed during each iteration.

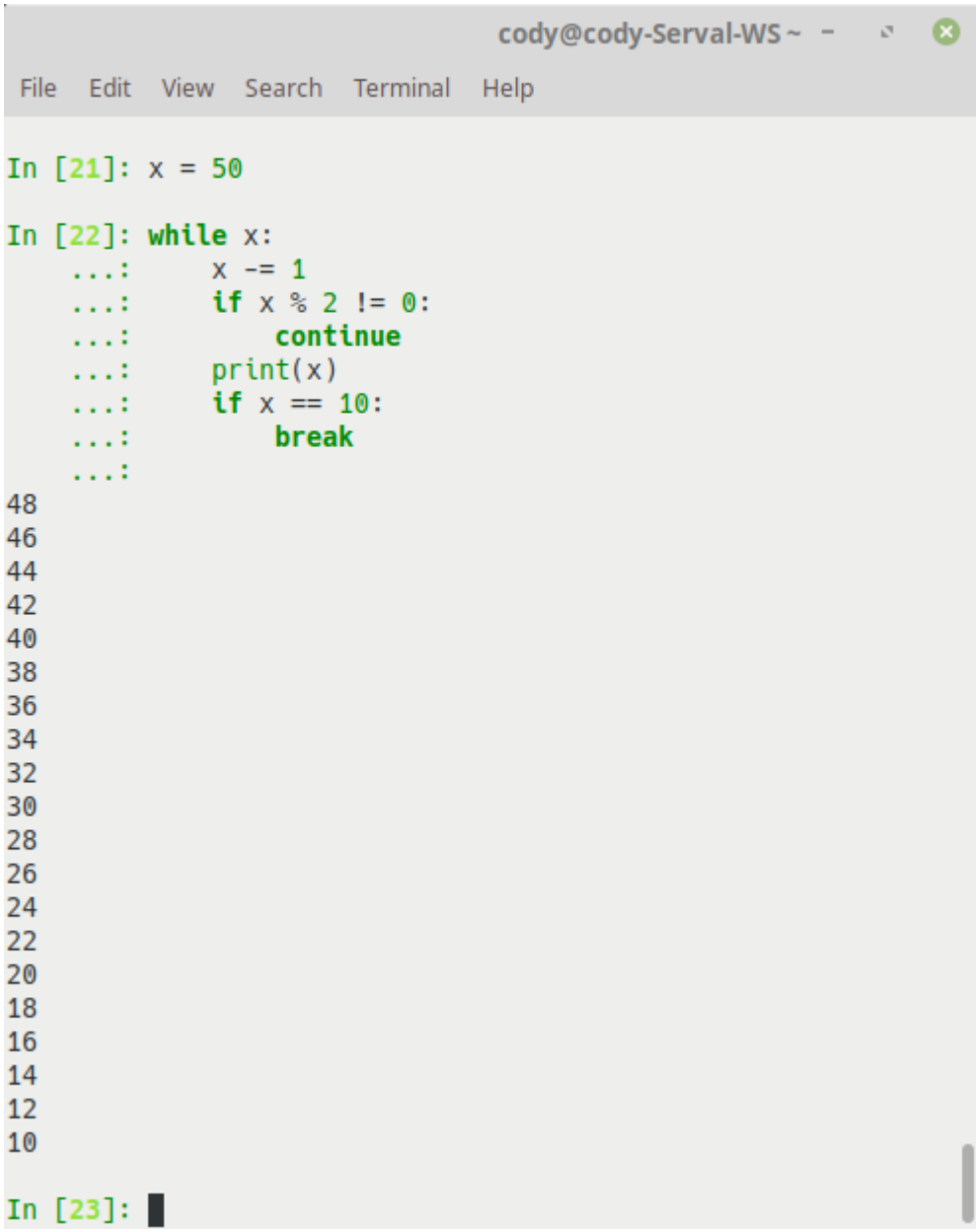
Line 3 tests a particular condition, usually whether the initial condition is still true. If line 3 is a true statement, the loop quits. If not, the control moves to line 5, which is yet another test. In this case, if the test condition is true, the loop immediately starts again and ignores the following logic.

Line 8 is executed when the looping condition is no longer true.

Following screenshot shows a simple `while` loop that loops 10 times before the test condition causes it to stop. When the loop is finished, the `else` statement ensures a final operation is performed, in this case printing a string:


IPython: home/cody
File Edit View Search Terminal Help
In [1]: i = 0
In [2]: while i < 10:
...: print("{} is less than 10.".format(i))
...: i += 1
...: else:
...: print("{} is equal to 10.".format(i))
...:
0 is less than 10.
1 is less than 10.
2 is less than 10.
3 is less than 10.
4 is less than 10.
5 is less than 10.
6 is less than 10.
7 is less than 10.
8 is less than 10.
9 is less than 10.
10 is equal to 10.
In [3]:
Simple while loop

Following screenshot shows a more complex loop. In this case, it counts down from `50`, printing all even numbers. Once the number `10` is reached, the loop quits. The `%` symbol is the modulus operator and is explained later in this book:



More complex while loop

for loops

We've seen `for` loops in previous examples; they are the go-to sequence iterator for Python. The `for` loops work on nearly anything: strings, lists, tuples, and so on. The main format is shown next. Note how `for` loops have the same basic look as `while` loops:

```
for <target> in <object>:  # assign object items to target
    <statements>
    if <test>:
        break  # exit loop now, skip else
    if <test>:
        continue  # go to top of loop now
    else:
        <statements>  # if we didn't hit a 'break'
```

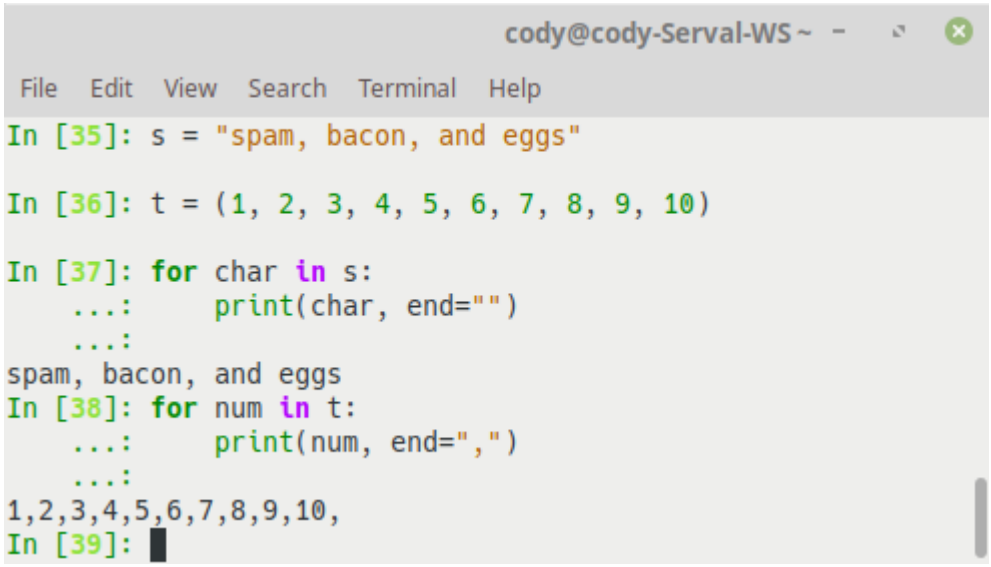
When the `for` loop starts, it looks at the first item in the sequence. This item is given a value of `0` (many programming languages start counting at 0, rather than 1). Once the code block has finished doing its processing, the `for` loop looks at the second value and gives it a value of `1`. Again, the code block does its processing and the `for` loop looks at the next value and gives it a value of `2`. This sequence continues until there are no more values in the list. At that point, the `for` loop stops and the control proceeds to the next operation in the program. Following screenshot shows how nested `for` loops are:

cody@cody-Serval-WS ~
File Edit View Search Terminal Help
In [27]: for i in range(2, 20):
...: for x in range(2, i):
...: if i % x == 0:
...: print("{} equals {} * {}".format(i, x, i/x))
...: break
...: else:
...: print("{} is a prime number".format(i))
...:
...:
2 is a prime number
3 is a prime number
4 equals 2 * 2.0
5 is a prime number
6 equals 2 * 3.0
7 is a prime number
8 equals 2 * 4.0
9 equals 3 * 3.0
10 equals 2 * 5.0
11 is a prime number
12 equals 2 * 6.0
13 is a prime number
14 equals 2 * 7.0
15 equals 3 * 5.0
16 equals 2 * 8.0
17 is a prime number
18 equals 2 * 9.0
19 is a prime number
In [28]:
for loop

The first loop increments a counter from `2` through `20`, while the second loop increments from `2` through the current count. If the remainder of `i` divided by `x` is `0`, the inner loop breaks and returns to the outer loop.

The `%` symbol is the modulus operation, which returns the remainder of a division operation.
The `range()` function automatically generates a list of integers, in memory, starting from the first number provided (inclusive) and stopping at the second number provided (exclusive). A third number can be provided that indicates how many numbers to skip between each integer generated.

Strings and tuples can also be a good location to use a `for` loop. Iteration through the items in a string or tuple is easily accomplished with a loop, and is frequently used for processing text. Following screenshot demonstrates the sequence iteration:


cody@cody-Serval-WS ~
File Edit View Search Terminal Help
In [35]: s = "spam, bacon, and eggs"
In [36]: t = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
In [37]: for char in s:
...: print(char, end="")
...:
spam, bacon, and eggs
In [38]: for num in t:
...: print(num, end=",")
...:
1,2,3,4,5,6,7,8,9,10,
In [39]:
for loops with sequences

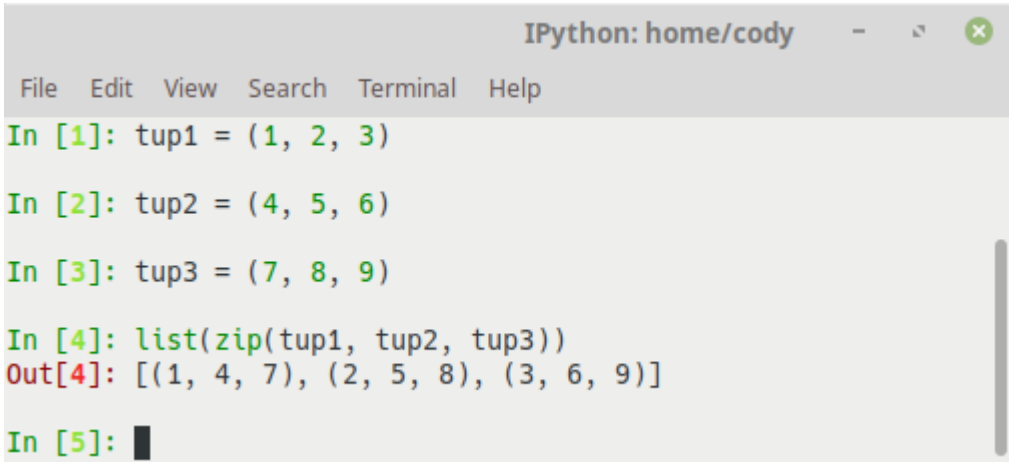
A string and a tuple are defined in lines 35 and 36, respectively. In line 37, the `for` loop iterates over, and prints out the string, character by character. In line 38, a similar thing occurs with the tuple, except we have told the `print()` function to add a comma to the end of each number printed. Note that a comma is included after the final number due to this feature.

Because `for` loops are pretty simple and tend to run more quickly than `while` loops, they are the preferred loop when iterating through a sequence. In general, avoid counting things in Python, since the built-in iteration tools can do much of the work you would have to manually write.

zip() function

The `zip()` function is a great tool to process multiple sequences during the same loop. As it looks at each sequence, `zip()` joins the individual items from the sequences and outputs tuples with parallel items from each sequence.

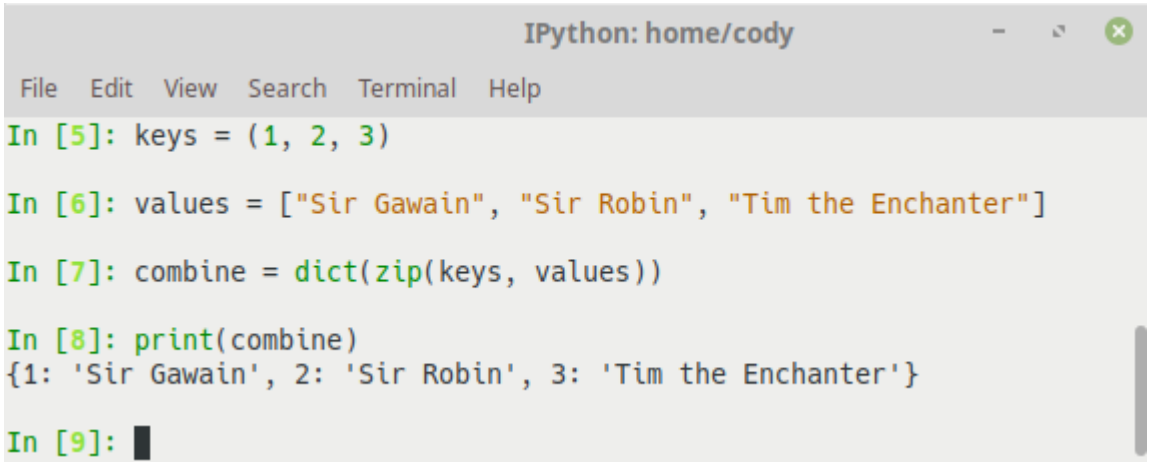
In other words, if you stack two rows of items on top of each other, `zip()` will gather the first two items from each row and combine them in a tuple. Then, it gets the second two items from each row and makes another tuple. It continues doing this until each row is finished. Following screenshot demonstrates this functionality:


IPython: home/cody
File Edit View Search Terminal Help
In [1]: tup1 = (1, 2, 3)
In [2]: tup2 = (4, 5, 6)
In [3]: tup3 = (7, 8, 9)
In [4]: list(zip(tup1, tup2, tup3))
Out[4]: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
In [5]:
zip() combines tuples

Each element in each tuple is combined concurrently as the list is generated, as `zip()` simply grabs the item in each tuple that has the same index value, and then makes a new tuple from them.

Any type of iterable sequence can be used by `zip()`, even files (as they are just streams of characters). Also, it will only process the minimum number of items from all given sequences; if you have a list with three items and another with seven, you'll only get three tuples as the result.

The `zip()` function is great for combining data gathered at runtime, such as user input or from a file. If you need to make a dictionary after the program has been started, because you don't know what the actual dictionary values will be, `zip()` can help create that dictionary. While following screenshot demonstrates this, the keys and values are frequently from sources created after the program was started:


IPython: home/cody
File Edit View Search Terminal Help
In [5]: keys = (1, 2, 3)
In [6]: values = ["Sir Gawain", "Sir Robin", "Tim the Enchanter"]
In [7]: combine = dict(zip(keys, values))
In [8]: print(combine)
{1: 'Sir Gawain', 2: 'Sir Robin', 3: 'Tim the Enchanter'}
In [9]:
Dictionary through zip() function

Line 7 shows how the Python built-in constructor `dict()` can be used to make an object creation request. It essentially makes a dictionary out of a list, and can be quite powerful when working with dictionaries. There are other constructors within Python, such as lists, and they are often used to programmatically create these objects during runtime.

Exceptions

Exceptions are events that modify a program's flow, either intentionally or due to errors. Some examples are trying to open a file that doesn't exist, or when the program reaches a marker, such as the completion of a loop. Exceptions, by definition, don't occur very often; hence, they are the **exceptions to the rule** and a special class has been created for them.

Exceptions are everywhere in Python. Virtually every module in the Python standard library uses them, and Python itself will raise them in a variety of different circumstances. Here are just a few examples:

- Accessing a non-existent dictionary key will raise a `KeyError` exception
- Searching a list for a non-existent value will raise a `ValueError` exception
- Calling a non-existent method of a class will raise an `AttributeError` exception
- Referencing a non-existent variable will raise a `NameError` exception
- Mixing data types without coercion will raise a `TypeError` exception

One use of exceptions is to catch a fault and allow the program to continue working; this is the most common way to use exceptions. If the developer can anticipate possible errors, then the exception-catching code can be written to deal with them. This not only allows the program to continue working, but also hides the ugliness of broken software from the user. From a security perspective, it also helps make it more difficult to screenshot out what a program does if someone is looking for vulnerabilities to exploit.

When programming with the Python command-line interpreter, you generally don't need to worry about catching exceptions. Your program is usually short enough to not be hurt too much if an exception occurs. Plus, having the exception occur at the command-line is a quick and easy way to tell if your code logic has a problem. However, if the same error occurred in your real program, it would fail and stop working.

Exceptions can be created manually in the code by *raising* an exception. It operates exactly as a system-caused exception, except that the programmer is doing it on purpose for a number of reasons. One of the benefits of using exceptions is that, by their nature, they don't put any overhead on the code processing. Because exceptions aren't supposed to happen very often, they aren't processed until they occur.

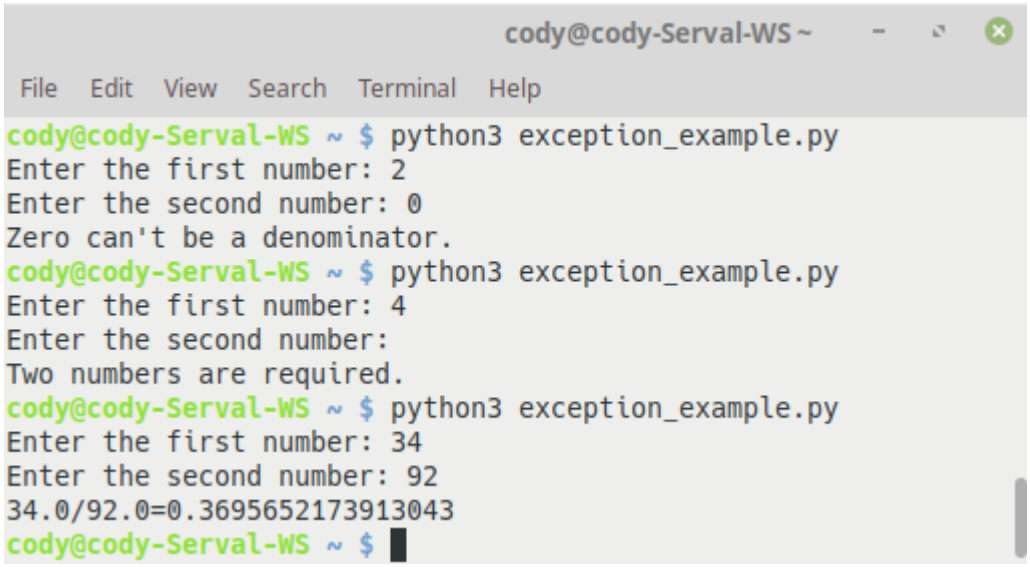
Exceptions can be thought of as a special form of the `if...else` statements. You can realistically do the same thing with `if` blocks as you can with exceptions. However, as already mentioned, exceptions aren't processed until they occur; `if` blocks are processed all the time. Proper use of exceptions can help the performance of your program. The more infrequent the error might occur, the better off you are to use exceptions; using `if` blocks requires Python to always test extra conditions before continuing. Exceptions also make code management easier: if your programming logic is mixed in with error-handling `if` statements, it can be difficult to read, modify, and debug your program.

The following is a simple program that highlights most of the important features of exception processing:

```
1 num1 = input("Enter the first number: ")
2 num2 = input("Enter the second number: ")
3 try:
4    num1 = float(num1)
5    num2 = float(num2)
6    result = num1/num2
```

```
except ValueError:
   print ("Two numbers are required.")
except ZeroDivisionError:
   print ("Zero can't be a denominator.")
else:
   print ("{num1}/{num2}={result}".format(num1=num1, num2=num2, result=result))
```

This program simply takes two numbers from the user, divides them, and then shows the calculation and result to the user. If the user forgets to enter a number or attempts to divide by zero, the resultant errors are caught and the user is politely informed of the problem. Following screenshot shows the output of this program:


cody@cody-Serval-WS ~
File Edit View Search Terminal Help
cody@cody-Serval-WS ~ $ python3 exception_example.py
Enter the first number: 2
Enter the second number: 0
Zero can't be a denominator.
cody@cody-Serval-WS ~ $ python3 exception_example.py
Enter the first number: 4
Enter the second number:
Two numbers are required.
cody@cody-Serval-WS ~ $ python3 exception_example.py
Enter the first number: 34
Enter the second number: 92
34.0/92.0=0.3695652173913043
cody@cody-Serval-WS ~ $
Exception example output

As demonstrated in the code, you can catch multiple exceptions within the same `try` block. You can also put an optional `else` statement at the end to denote the logic to perform if all goes well. Exceptions assume the "default" case to be true until an exception actually occurs. This speeds up program processing, compared to `if...else` statements that check each `if` line for a true or false condition.

One change you could make to this program is to simply put it all within the `try` block. The `input()` variables (which capture input from the user's keyboard) could be placed within the `try`block, replacing the `num1` and `num2` variables by forcing the user input to a float value, as shown here:

```
try:
    numerator = float(input("Enter the numerator."))
    denominator = float(input("Enter the denominator."))
```

This way, you reduce the amount of logic that has to be written, processed, and tested. You still have the same exceptions; you're just simplifying the program.

If you want to ensure some actions take place at the end of the `try` block, you can add a `finally`block at the end. While `try`/`except` is used to catch and recover from errors, `try`/`finally` is used to ensure that some sort of action takes place, regardless of whether any exceptions actually occur, such as closing files, severing server connections, and so on.

When planning your project, it's better to include error-checking, such as exceptions, in your code as you program rather than as an afterthought. A special "category" of programming involves writing test cases to ensure that most possible errors are accounted for in the code, especially as the code changes or new versions are created. By planning ahead and putting exceptions and other error-checking code into your program at the outset, you ensure that problems are caught before they can cause problems. By updating your test cases as your program evolves, you ensure that version upgrades maintain compatibility and that a fix doesn't create an error condition.

Exception class hierarchy

When an exception occurs, it starts at the innermost level possible (a child) and travels upward (through the parents), waiting to be caught. This means a couple of things to a programmer:

- If you don't know what exception may occur, you can always just catch a higher-level exception. For example, if you didn't know that `ZeroDivisionError` from the preceding exception example was its own exception, you could have used the `ArithmeticError` for the exception and caught that. The following exceptions list shows that `ZeroDivisionError` is a child of `ArithmeticError`, which in turn is a child of `Exception`, which in turn is a child of `BaseException`, which is the default class that all other exceptions derive from.
- Multiple exceptions can be treated the same way. Following on with the preceding example, suppose you plan on using `ZeroDivisionError` and you want to include `FloatingPointError`. If you wanted to have the same action taken for both errors, simply catch the parent exception, `ArithmeticError`. That way, when either a floating-point or a zero-division error occurs, you don't have to have a separate case for each one. Naturally, if you have a need or desire to catch each one separately, perhaps because you want different actions to be taken, then writing exceptions for each case is fine. Also, catching the parent exception means all children exceptions of that parent, even ones you didn't intend to catch.

The following exceptions list shows the hierarchy of exceptions from the **Python Library Reference**, for Python 3.6. One thing to note is that this hierarchy sometimes changes between Python versions; for example, `ArithmeticError` is a child of `StandardError` in version 2.7, but is a child of `Exception` in version 3.6:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
```

```
|    +-- KeyError
+-- MemoryError
+-- NameError
|    +-- UnboundLocalError
+-- OSError
|    +-- BlockingIOError
|    +-- ChildProcessError
|    +-- ConnectionError
|    |    +-- BrokenPipeError
|    |    +-- ConnectionAbortedError
|    |    +-- ConnectionRefusedError
|    |    +-- ConnectionResetError
|    +-- FileExistsError
|    +-- FileNotFoundError
|    +-- InterruptedError
|    +-- IsADirectoryError
|    +-- NotADirectoryError
|    +-- PermissionError
|    +-- ProcessLookupError
|    +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|    +-- NotImplementedError
|    +-- RecursionError
+-- SyntaxError
|    +-- IndentationError
|         +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|         +-- UnicodeDecodeError
|         +-- UnicodeEncodeError
|         +-- UnicodeTranslateError
+-- Warning
     +-- DeprecationWarning
     +-- PendingDeprecationWarning
     +-- RuntimeWarning
     +-- SyntaxWarning
     +-- UserWarning
     +-- FutureWarning
     +-- ImportWarning
     +-- UnicodeWarning
     +-- BytesWarning
     +-- ResourceWarning
```

This hierarchy is important to understand when you're writing your code; an exception may not be available depending on which Python version you're using. If you're trying to make your programs as portable as possible, you have to be cognizant of which Python version the target system is running.

User-defined exceptions

Python allows for a programmer to create new exceptions if the project warrants them. However, make sure that one of the built-in exceptions won't do the job for you. They have been tested and tweaked over the years and not only do they work effectively, but they have been optimized for performance and are bug-free.

Making your own exceptions involves object-oriented programming, which will be covered in `Chapter 4`, *Functions and Object Oriented Programming,* in the *Classes, methods, and namespaces* section. To make a custom exception, the programmer determines which base exception to use as the class to inherit from. For example, making an exception for negative numbers or one for imaginary numbers would probably fall under the `ArithmeticError` exception class.

To make a custom exception, simply inherit the base exception and define what it will do. The following example gives an example of creating a custom exception:

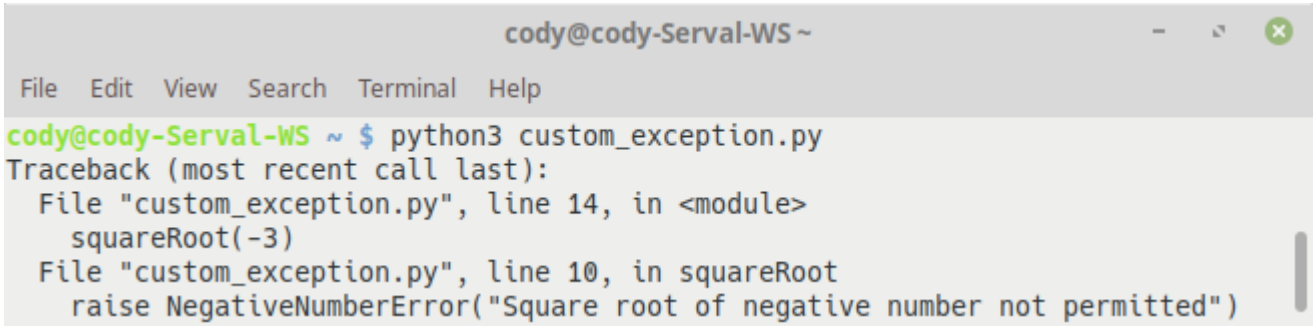
```
1  import math
2
3  class NegativeNumberError(ValueError):
4      """Attempted improper operation on negative number."""
5      pass
6
7  def squareRoot(number):
8      """Computes square root of number. Raises NegativeNumberError if number is less
than 0."""
9      if number < 0:
10         raise NegativeNumberError("Square root of negative number not permitted")
11
12     return math.sqrt(number)
13
14 if __name__ == "__main__":
15     squareRoot(-3)
```

Because this is dealing with square roots, we need to import the `math` module in line 1. We inherit from `ValueError` in line 3, creating the new class, `NegativeNumberError`. Because we are inheriting all the characteristics of `ValueError`, we don't have to do anything else, so we just use `pass` in line 5 to tell Python to continue with the program.

To actually use the new exception, a function is created in lines 7-12 that calls `NegativeNumberError`if the argument value is less than `0`; otherwise, it gives the square root of the number.

Line 14 tells Python to run everything following if this program is explicitly called; that is, if this is the main program to be processed. Line 15 is just a test to ensure that the new exception works as expected.

Following screenshot shows the output of this program, showing that the new exception works as advertised:


cody@cody-Serval-WS ~
File Edit View Search Terminal Help
cody@cody-Serval-WS ~ $ python3 custom_exception.py
Traceback (most recent call last):
File "custom_exception.py", line 14, in <module>
squareRoot(-3)
File "custom_exception.py", line 10, in squareRoot
raise NegativeNumberError("Square root of negative number not permitted")
Custom exception output

Final thoughts

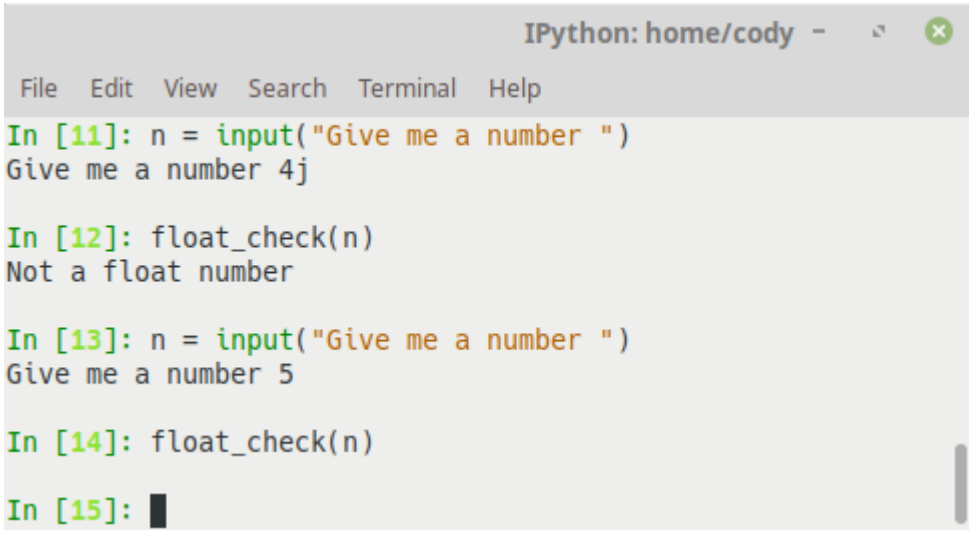
It's important to reiterate that exceptions don't always have to resolve errors. Since they function like `if...else` blocks, they can be used in a similar manner. For example, the **end-of-file exception** (**EOFError**) can be used to identify when the last line of a file has been processed, thereby breaking out of the processing loop.

One possible use of this is to analyze strings. If a string method won't work for you, you can generally accomplish the same goal using exceptions. For example, if you want to determine if a string value is a float, you can't use the `str.isdigit()` method to do the job; it can only tell you if a particular string entry matches the values `0-9`. If you have a floating-point number, which includes a decimal point, the `isdigit()` method gives you an error.

In this case, you can simply check whether the value is a float and expressly check for the error, as demonstrated here:

```
1 def float_check(num):
2   try:
3     float(num)
4   except ValueError:
5     print("Not a float number.")
```

In this example, if you check a string and it doesn't match a floating-point number, or can't be converted into a float, you receive `ValueError`, alerting you to the fact that the value is not a float, as shown in following screenshot:


IPython: home/cody
File Edit View Search Terminal Help
In [11]: n = input("Give me a number ")
Give me a number 4j
In [12]: float_check(n)
Not a float number
In [13]: n = input("Give me a number ")
Give me a number 5
In [14]: float_check(n)
In [15]:
Float check output

When using `input()` to receive captured information from the user, the information received is saved as a string. In this case, the input for line 11 was an imaginary number, which can't be converted to a float value and the error message was printed in line 12. But when an integer is provided in line 13, Python can convert it to a floating-point number and no error is generated.

An easier way to perform this check, without using exceptions, is shown here:

```
if type(num) is not float:
    print("Not a float number.")
```

In other languages, normal programming practice is to use exceptions for exceptional cases: if something goes wrong, there is a serious problem; otherwise, continue normal operations. Python programming is different, where the assumption is that valid data is present and, if not, an exception is generated and caught. In the end, the choice is up to you; use what makes the most sense and is most readable.

Summary

In this chapter, we covered the basics of controlling programming logic. This included conditional branching using `if...else` statements, as well as repetition using `while` and `for` loops. Finally, we talked about error handling using exceptions, and how exceptions can substitute for `if...else` statements in certain cases.

In the next chapter, we will talk about the key parts of programs that allow for modularity: functions and classes. We will also look at how to use properties to provide the ability to get and set parameter values.

Functions and Object Oriented Programming

So far, we've covered a lot about how the Python language is structured, how to use the built-in data structures, and some basic example programs. Now, we'll dive into how to make more complex programs.

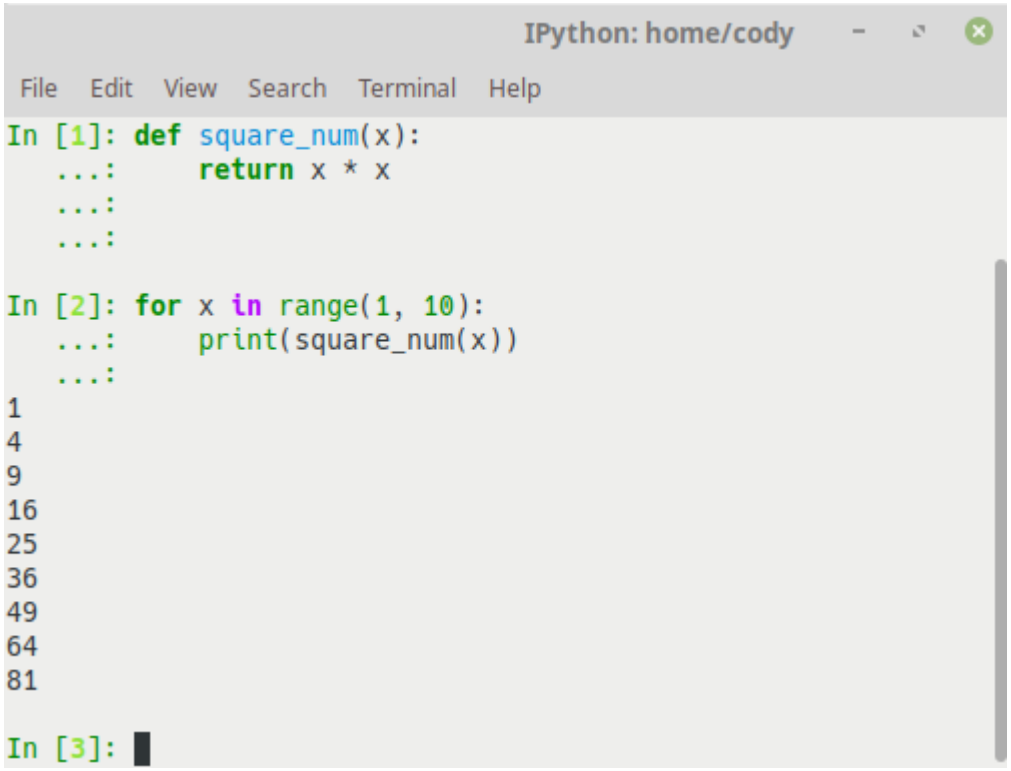
Most Python programs are built using a variety of variables and functions, classes, or both. Functions are programmer-created code blocks that do a specific task. Classes are object-oriented structures that we'll look at in the *Classes, methods, and namespaces* section; briefly, though, classes help segregate code into related code blocks and can include class-specific variables and methods (class-specified functions).

In this chapter, we will cover the following topics:

- Working with functions
- Classes, methods, and namespaces
- Properties and class and static methods

Working with functions

Functions are blocks of code logic that can be used multiple times within a program, simply by calling the function's name. If you don't use functions, but you want to repeat an operation multiple times, you will have to copy and paste the necessary code multiple times to get your program to work. If you ever have to revise the program, you would have to ensure that all applicable copied and pasted code is updated. With functions, a single code block can be updated. The following is an example of a function:



Function example

Preceding screenshot is about as simple as it gets. On line 1, we use the `def` keyword to define the function called `square_num()` (the parentheses indicate that it's a function rather than a statement) and tell it that the argument called `x` will be used for processing. Then we actually define what the function will do; in this case, it will multiply `x` by itself to produce a squared value. By using the `return` statement, the squared value will be returned back to whatever actually called the function (in this case, the `print()` function on line 2).

Next, on line 2, we create a `for` loop that iterates over a range of numbers, from `1` to `9`; for each number, we call the `square_num()` function and then print the resultant squared value.

While this particular example was generated interactively in IPython, if you wanted to save this program to a file for later use, you would simply type the program into a text file, such as the following example, `function_example.py`:

```
def square_num(x):
    return x * x

for x in range(1, 10):
    print(square_num(x))
```

To run the program, you first have to save it to your computer and give it a name ending in `.py`, for example, `function_example.py`. Then, on the command line, type `python3 function_example.py`. The results should look like following screenshot:

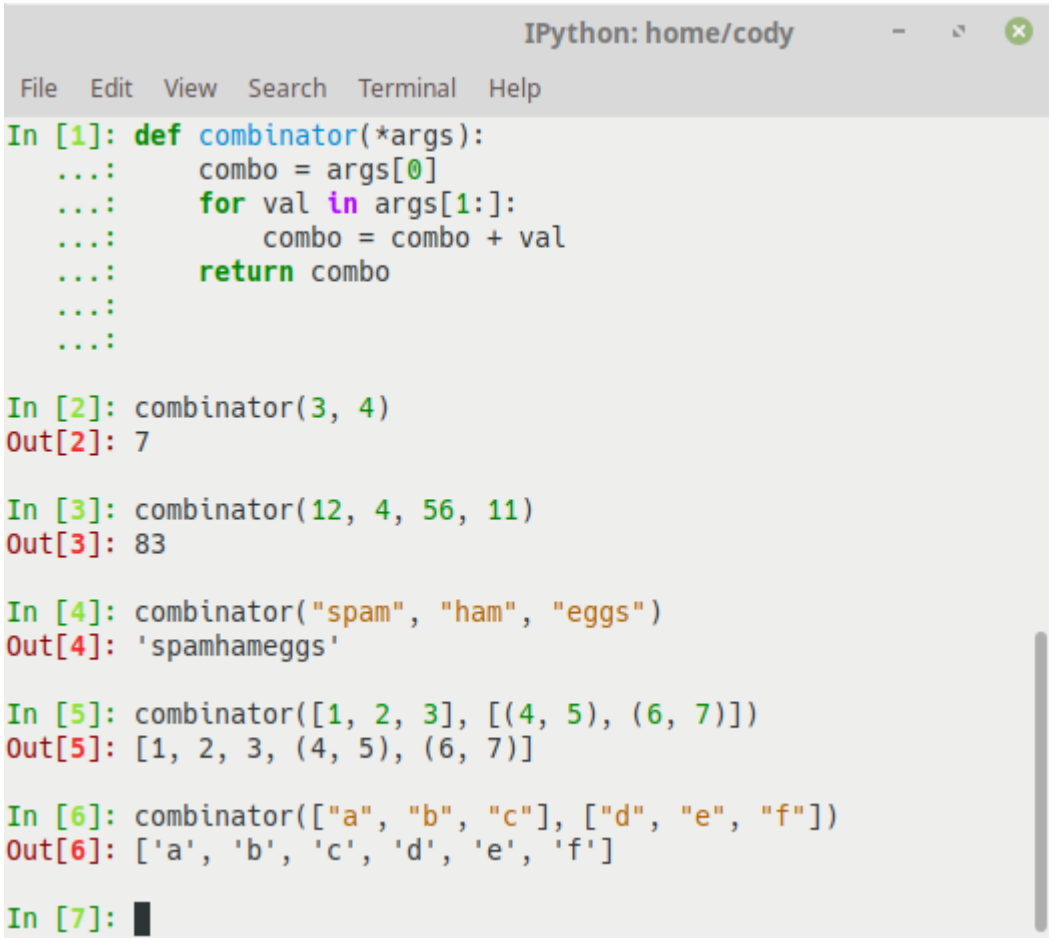
function_example.py output

Next we will talk about a neat little trick that demonstrates the power of Python: polymorphism. We touched on it in `Chapter 2`, *Data Types and Modules*, in the *Basic string operations* section, but we'll discuss it in more depth here, as functions are one way to create polymorphism in your program.

The code in `function_example.py` shows that Python is capable of interpreting (to an extent) what you want to do. In this case, the `*` operator is overloaded to support multiple roles. If numbers are provided, it will multiply them; if a string and number is provided, it will perform repetition; and so on.

As long as the objects passed into a function support the intended action (as determined by the Python language), the function will process them. If a particular operation can't be performed on the objects passed in, the interpreter will return an error, letting the programmer know of the problem.

This feature is not generally found in static typed languages, such as C/C++. This is because static languages can only deal with data types that are explicitly stated; attempting to override the behavior won't work. You would have to create a function to perform repetition if you wanted to use the `*` operator, as it is used for number multiplication only. An example of polymorphism is displayed in following screenshot:


IPython: home/cody
File Edit View Search Terminal Help
In [1]: def combinator(*args):
...: combo = args[0]
...: for val in args[1:]:
...: combo = combo + val
...: return combo
...:
...:
In [2]: combinator(3, 4)
Out[2]: 7
In [3]: combinator(12, 4, 56, 11)
Out[3]: 83
In [4]: combinator("spam", "ham", "eggs")
Out[4]: 'spamhameggs'
In [5]: combinator([1, 2, 3], [(4, 5), (6, 7)])
Out[5]: [1, 2, 3, (4, 5), (6, 7)]
In [6]: combinator(["a", "b", "c"], ["d", "e", "f"])
Out[6]: ['a', 'b', 'c', 'd', 'e', 'f']
In [7]:
Polymorphism example

This little program is pretty powerful, as it takes a variable number of arguments and either adds them or concatenates (combines) them together, depending on the argument type. These arguments can be anything: numbers, strings, lists, tuples, and so on.

We mentioned the `*args` keyword previously in Chapter 2, *Data Types and Modules*, in the *String methods* section. This is a special feature of Python that allows you to enter undesignated arguments and do things to them (such as add them together). The `*` is like a wildcard; it signifies that a variable number of arguments can be provided. You could get by without using the word `args`, but common practice is to include it for clarity.

A similar argument keyword is `**kwargs`. This one is related (it takes an unlimited number of arguments), but the arguments are set off by keywords.

This way, you can match variables to the arguments based on the keywords. `*args` and `**kwargs`can be used together, if desired.

Lambdas

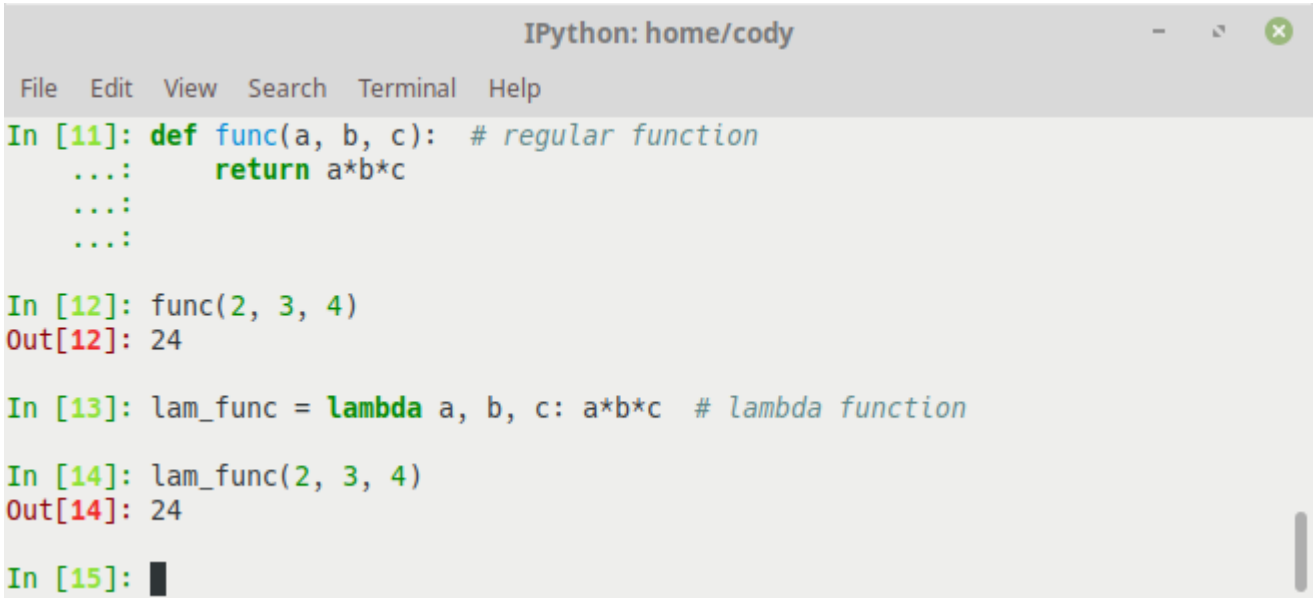
An interesting way of making functions is through anonymous functions, also known as lambda expressions. Lambdas create a function that is essentially processed in-line, rather than giving it a name to be called later (hence, the anonymous part). This is useful when the normal `def` statement wouldn't necessarily work, such as within a list or as a function call's argument.

When using the normal `def` statement to create a function, the function is assigned a name and called from a different location, possibly multiple times; it also returns an object of some type. When using a lambda, the function itself is the returned object that can be given a name, if desired.

Lambdas look and work differently than regular functions. First, it is a single expression, not a block of code. This is why it can be used in places where a normal function can't; you're essentially creating the `return` statement directly, rather than calling a function to return a value.

Second, because it's an expression, there's a limit to what it can do. Since expressions break down to a value, whereas statements perform code logic, lambdas are limited in how much work they can do. If you need to do significant work, such as working with `if` statements or loops, you'll have to create a normally defined function.

Following screenshot provides an example of a normal function versus its equivalent lambda function:


IPython: home/cody
File Edit View Search Terminal Help
In [11]: def func(a, b, c): # regular function
    ...:   return a*b*c
    ...:
    ...:
In [12]: func(2, 3, 4)
Out[12]: 24
In [13]: lam_func = lambda a, b, c: a*b*c # lambda function
In [14]: lam_func(2, 3, 4)
Out[14]: 24
In [15]:
Lambda function

It's somewhat confusing at first, but, as you can see, the lambda is doing the exact same thing that the traditional function does, except it assigns the return value to a variable name automatically, rather than having to be called separately later. The lambda shortcuts the function process in a similar manner as list comprehensions shortcut list creation.

Lambdas work the same as normal functions, with all the features and limitations (actually, a few more limitations, since they only use expressions). Lambdas are a nice feature when you only need quick, inline

executable code and a normal function won't work. You'll also find some third-party libraries that require a lambda expression as an argument.

Classes, methods, and namespaces

Object-oriented programming (**OOP**) is one of the primary ways to program in many languages, and sometimes the only way. Classes are the primary way OOP is implemented in most object-oriented languages, and Python uses classes as well. While functions are great and can do a lot of work, you'll see that OOP is very powerful as well. Plus, many Python libraries and APIs use classes, so you should at least be able to understand what the code is doing.

One thing to note about Python and OOP is that it's not mandatory to use classes in your code. As you've already seen, Python can do just fine with functions. Unlike languages such as Java, you aren't tied down to a single way of doing things; you can mix functions and classes as necessary in the same program. This lets you build the code in a way that works best; maybe you don't need to have a full-blown class with initialization code and methods to just return a calculation. With Python, you can get as technical as you want.

How are classes better?

Imagine you have a program that calculates the velocity of a car in a two-dimensional plane using functions. If you want to make a new program that calculates the velocity of an airplane in three dimensions, you can use the concepts of your car functions to make the airplane model work, but you'll have to rewrite many of the functions to make them work for the vertical dimension, especially to map the object in a 3D space. You may be lucky and be able to copy and paste some of them, but for the most part, you'll have to redo much of the work.

Classes let you define an object once, and then reuse it multiple times. You can give it a base function (functions in classes are called **methods** to indicate that they are object-oriented) and then build upon that method to redefine it as necessary. It also lets you model real-world objects much better than using functions. In short, a class provides the basic template and default behavior for an object, and an instance of that class, that is, a particular incarnation created from the class, uses the base template as a foundation for more specific changes.

For example, you could make a tire class that defines the size of the tire, how much pressure it holds, what it's made of, and so on, and then make methods to determine how quickly it wears down based on certain conditions. You can then use this tire class as part of a car class, a bicycle class, or whatever. Each use of the tire class (called instances) would use different properties of the base tire object. If the base tire object said it was just made of rubber, perhaps the car class would "enhance" the tire by saying it had

steel bands or maybe the bike class would say it had an internal air bladder. This will make more sense later.

Several concepts of classes are important to know:

- Classes have a definite namespace, just like modules. Trying to call a class method from a different class will give you an error unless you qualify it using the dot protocol; for example, `spamClass.eggMethod()`. We have seen this when a module is imported, then we attempt to use a method from inside it. As we saw in `Chapter 2`, *Data Types and Modules,* in the *Types of imports* section, for example, if you simply `import math`, you have to qualify anything that comes from it, such as `math.log()`. However, if you use `from math import *`, then you don't have to qualify it, as all those items are now part of your program's namespace. Of course, if you happen to make your own variables or methods with the same names, you'll run into problems due to variable shadowing.
- Classes support multiple copies. This is because classes have two different object types: class objects and instance objects. Class objects give the default behavior and are used to create instance objects. Instance objects are the objects that actually do the work in your program; every time a class is called, a new instance is created. You can have as many instance objects of the same class object as you need. Instance objects are normally marked by the `self` keyword, so a class method could be `Car.Brake()`, while a specific instance of the `Brake()` method would be marked as `self.Brake()`. (This is covered in more depth in `Chapter 4`, *Functions and Object Oriented Programming,* in the *Classes and Instances* section.)
- Each instance object has its own namespace but also inherits from the base class object. This means that each instance has the same default namespace components as the class object, but additionally, each instance can make new namespace objects just for itself. This is part of each instance being a new and separate object. In other words, an instance has objects that are part of the instance scope, but it also has access to the base class' scope.
- Class objects are similar to any built-in variable in Python. So a list of Python objects can be sliced, indexed, concatenated, and so on, just like strings, lists, and other standard Python types. This is because everything in Python is actually a class object; we aren't actually doing anything new with classes, we're just learning how to better use the inherent nature of the Python language.

Here's a brief list of Python OOP concepts:

- The `class` statement creates a class object and gives it a name, as well as creating a new namespace.
- Variable assignments and methods within the class create class `attributes`. These attributes are accessed by qualifying the name using dot syntax: `ClassName.Attribute`.
- Class attributes export the state of an object and its associated behavior. These attributes are shared by all instances of a class.
- Calling a class creates a new `instance` of the class. This is where the multiple copies part comes in.
- Each instance gets ("inherits") the default attributes of its class, while also getting its own namespace. This prevents instance objects from overlapping and confusing the program.
- Using the term `self` identifies a particular instance, allowing for per-instance attributes. This allows items such as variables to be associated with a particular instance.

Classes and instances

Let's look at an example of creating a simple class and methods, plus some instances of the class:

IPython: home/cody
File Edit View Search Terminal Help
In [22]: class Knight:
...: def setName(self, name):
...: self.name = name
...: def display(self):
...: print(self.name)
...:
In [23]: x = Knight()
In [24]: y = Knight()
In [25]: z = Knight()
In [26]: x.setName("Sir Lancelot, the Brave")
In [27]: y.setName("Sir Galahad, the Pure")
In [28]: z.setName("Sir Robin, the Not-Quite-So-Brave-As-Sir-Lancelot")
In [29]: x.display()
Sir Lancelot, the Brave
In [30]: y.display()
Sir Galahad, the Pure
In [31]: z.display()
Sir Robin, the Not-Quite-So-Brave-As-Sir-Lancelot
In [32]: x.name = "Sir Not-Appearing-In-This-Film"
In [33]: x.display()
Sir Not-Appearing-In-This-Film
Class and instances

The class is defined on line 22. We use the statement `class`, followed by the name of the class. Normally, when the class object is defined, there are no parentheses at the end; parentheses are only used for functions and methods. However, the old way of defining classes used parentheses, so you may see that in many programs you come across.

After the class is defined, two methods are defined as well: `setName()` and `display()`. The first argument in the parentheses for a method must be `self`. This is used to identify which particular instance is calling the method. The Python interpreter handles the calls internally, so all you have to do is make sure `self` is where it's supposed to be so you don't get an error.

Even though you must use `self` to identify each particular instance of a class, Python is smart enough to know which particular instance is being referenced, so having multiple instances at the same time is not a problem.

`self` is similar to `this`, which is used in several other languages, such as Java. Even if you have no arguments for a method, you must still include `self` so Python knows which class instance is being referred to, as demonstrated with the `display()` method.

If there are any arguments that are passed to a method, such as `name` in this case, they will follow the `self` argument. When you are assigning arguments or other objects to variables, such as `self.name`, the variable itself must be qualified with the "self" title. Again, this is used to identify a particular instance.

The two methods, `setName()` and `display()`, defined in the class simply accept a string argument and assign it to a "name" variable, and then print that name to the screen respectively.

Moving on, lines 23-25 create instances of the `Knight` class. Here, you'll notice that parentheses make an appearance. This is to signify that these are instance objects created from the `Knight`class. Each one of these instances has the exact same attributes, as they all inherit from the same parent class.

Lines 26-28 call the `setName()` method that is defined in the `Knight` class. However, each one is for a different instance: *x*, *y*, and *z* each has a different value for name. This can be seen on lines 29-31, which show the name that was set for each instance.

You can assign values to attributes in an instance during instance creation (lines 26-28) or by explicitly assigning to instance objects after creation (line 32). On line 32, we overwrite the name that was initially assigned to instance *x* by assigning a new name with the `=` sign. Generally speaking, you can do this with any instance at any time, assuming it's not a read-only parameter.

Modules

When working with modules, it's important to remember that classes can be imported directly from the module, rather than importing the entire module. As you learned in `Chapter 2`, *Data Types and Modules*, in the *Types of imports* section, `import` calls other libraries or modules, while `from` is a selective `import`, allowing you to get specific parts of a module; using `from foo import *` is dangerous, as it imports everything into the current namespace, which can cause shadowing of different objects, resulting in errors. Classes and functions are the tools that make importation work.

Since everything in Python is (from the interpreter's viewpoint) an object, everything can be imported. However, in practice, only classes and functions can be imported; if you have another program that is strictly in-line code, that is, no functions or classes, you'll just have to copy and paste it.

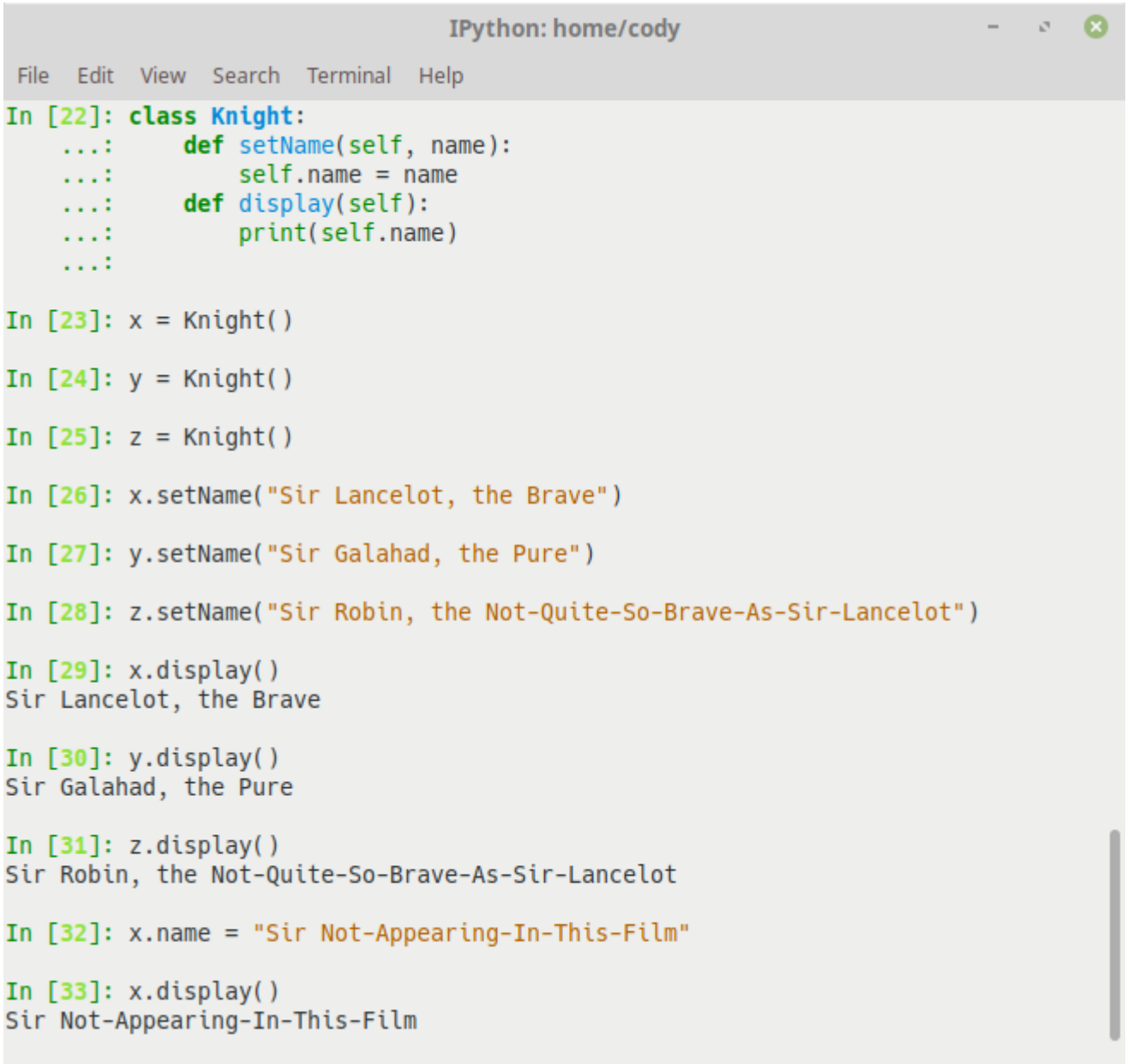
This is one reason why using classes or functions is important in your code. While you can make your code work without them, they improve the logic flow (since everything is segregated into its own logical block) and they make it possible to reuse code in different programs. Imagine trying to make a simulation program but you have to copy and paste code to screenshot out mathematical problems. It's easier to simply import the `math` library and utilize the classes available from it.

Inheritance

Inheritance is one of the most powerful aspects of classes. First off, classes allow you to modify a program without really making changes to it. To elaborate, by making a subclass through inheritance, you can change the behavior of the program by simply adding new components to it rather than rewriting the existing components.

As we've seen, an instance of a class inherits the attributes of that class. However, classes can also inherit attributes from other classes. Hence, a generic superclass (also known as a parent) can be specialized through subclasses (also called children), since the children classes inherit their general properties from the parent class. The subclasses can override the logic in a superclass, allowing you to change the behavior of subclasses without changing the superclass at all.

Let's look at a simple example, such as following screenshot:


IPython: home/cody
File Edit View Search Terminal Help
In [22]: class Knight:
...: def setName(self, name):
...: self.name = name
...: def display(self):
...: print(self.name)
...:
In [23]: x = Knight()
In [24]: y = Knight()
In [25]: z = Knight()
In [26]: x.setName("Sir Lancelot, the Brave")
In [27]: y.setName("Sir Galahad, the Pure")
In [28]: z.setName("Sir Robin, the Not-Quite-So-Brave-As-Sir-Lancelot")
In [29]: x.display()
Sir Lancelot, the Brave
In [30]: y.display()
Sir Galahad, the Pure
In [31]: z.display()
Sir Robin, the Not-Quite-So-Brave-As-Sir-Lancelot
In [32]: x.name = "Sir Not-Appearing-In-This-Film"
In [33]: x.display()
Sir Not-Appearing-In-This-Film

Class inheritance

First we make a class on line 43. This will be the superclass. This class has two methods, just like the preceding screenshot. Next, we make a subclass on line 44.

As you can see, the child class **overwrites** the display method but, since there is no explicit reference or redefining of the `set_data()` method, that particular method is inherited as is. When an instance of the `Parent` class is created, all of its actions will be taken from the methods defined in `Parent`. When a `Child` instance is created, it will use the inherited, generic `set_data()` method from `Parent` but the `display()` method will be the new one created in `Child`.

Instances of the superclass and subclass are created on lines 45 and 46. Both instances use the same `set_data()` method from `Parent`; `x` uses it because it's an instance of `Parent` while `y` uses it because `Child` inherits `set_data()` from `Parent`. However, when the `display()` method is called, `x` uses the definition from `Parent` (line 49) but `y` uses the definition from `Child` *(*line 50*)*, where display is overridden.

Because changes to program logic can be made through subclasses, the use of classes generally supports code reuse and extension better than traditional functions do. Functions have to be rewritten to change how they work, whereas classes can just be sub-classed to redefine methods.

It should be obvious now that a class instance can access all attributes from the classes they are part of; that is, a specific instance can access its own class methods, as well as any parent classes it is inherited from. Thus, you only have to overwrite or create specific attributes as necessary, relying on the inherited attributes to be available to the instance.

On a final note, you can use multiple inheritance (adding more than one superclass within the parentheses) if you need a class that belongs to different groups. In theory, this is good because it should cut down on extra work.

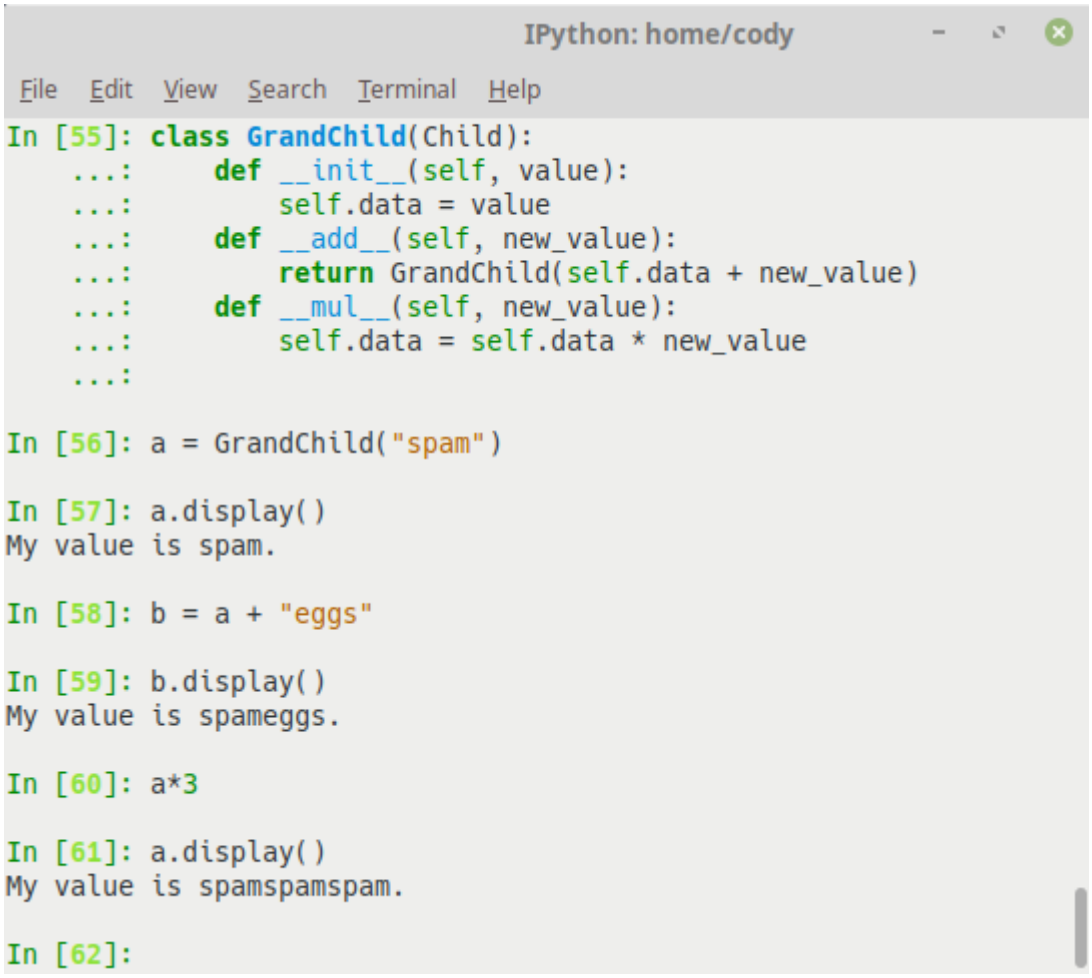
For example, a person could be a chef, a musician, a store owner, and a programmer; the person could inherit the properties from all of those roles. In reality, though, it can be a real pain to manage the multiple inheritance sets, as methods are inherited, overridden, and otherwise obfuscated. In short, multiple inheritance is something that beginning programmers should avoid until they understand how Python deals with it.

Operator overloading

Operator overloading was briefly talked about in `Chapter 2`, *Data Types and Modules*, in the *Basic string operations* section; now we will see how it works behind the scenes. Operator overloading simply means that objects that you create from classes can respond to actions (operations) that are already defined within Python, such as addition, slicing, printing, and so on. Even though these actions can be implemented through class methods, using overloading ties the behavior closer to Python's object model and the object interfaces are more consistent with Python's built-in objects; hence, overloading is easier to learn and use.

User-made classes can override nearly all of Python's built-in operation methods. These methods are identified by having two underscores before and after the method name, like this: `__add__`. These methods are automatically called when Python evaluates operators; if a user class overloads the `__add__` method, then, when an expression has the `+` symbol in it, the user's method will be used instead of Python's built-in method.

Continuing with the example from preceding screenshot, next screenshot shows how operator overloading would work in practice:



Creating an overloaded operator

We make a subclass of the `Child` class on line 55, so it is technically a sub-subclass of the original class, or a grandchild class. `GrandChild` doesn't override any of `Child`'s methods, so if you wanted, you could put the methods from `GrandChild` in `Child` and go from there. However, creating a new subclass allows you flexibility in your program.

When a new instance of `GrandChild` is made on line 56, the `__init__` method takes whatever argument is provided during instance creation and assigns it to the `self.data`. variable

`GrandChild` also overrides the `+` and `*` operators; when one of these is encountered in an expression, the object on the left of the operator is passed to the `self.data` argument and the object on the right is passed `new_value`, as demonstrated on lines 58-61. Note that the `Child`method of `display()` is utilized, as it was inherited and not modified.

These custom methods are different from the normal way Python deals with `+` and `*`, but these custom methods only apply to instances of `GrandChild`; other applications of these operators revert to the default Python functionality.

Some items of interest are shown in following screenshot:

Overloading problems

The `return` statement in the `__add__()` method creates a new `GrandChild` instance, with `self.data` and `new_value` automatically passed in as arguments. If this wasn't present, then line 59 would error out because instance `b` wasn't explicitly created, as happens on line 80.

On the other hand, attempting to perform an addition operation directly with instance *a*causes the memory location of the instance to be displayed (line 77), but the actual value of `a`never changed (line 78).

The `__mul__()` method doesn't return an instance, because it simply updates the value of *a* in-line (line 61). Thus, without the explicit creation of a new instance, attempting to assign `a*3` to variable `b` (line 79) causes the error on line 80, because `b` is just a regular variable and not an instance of `GrandChild`.

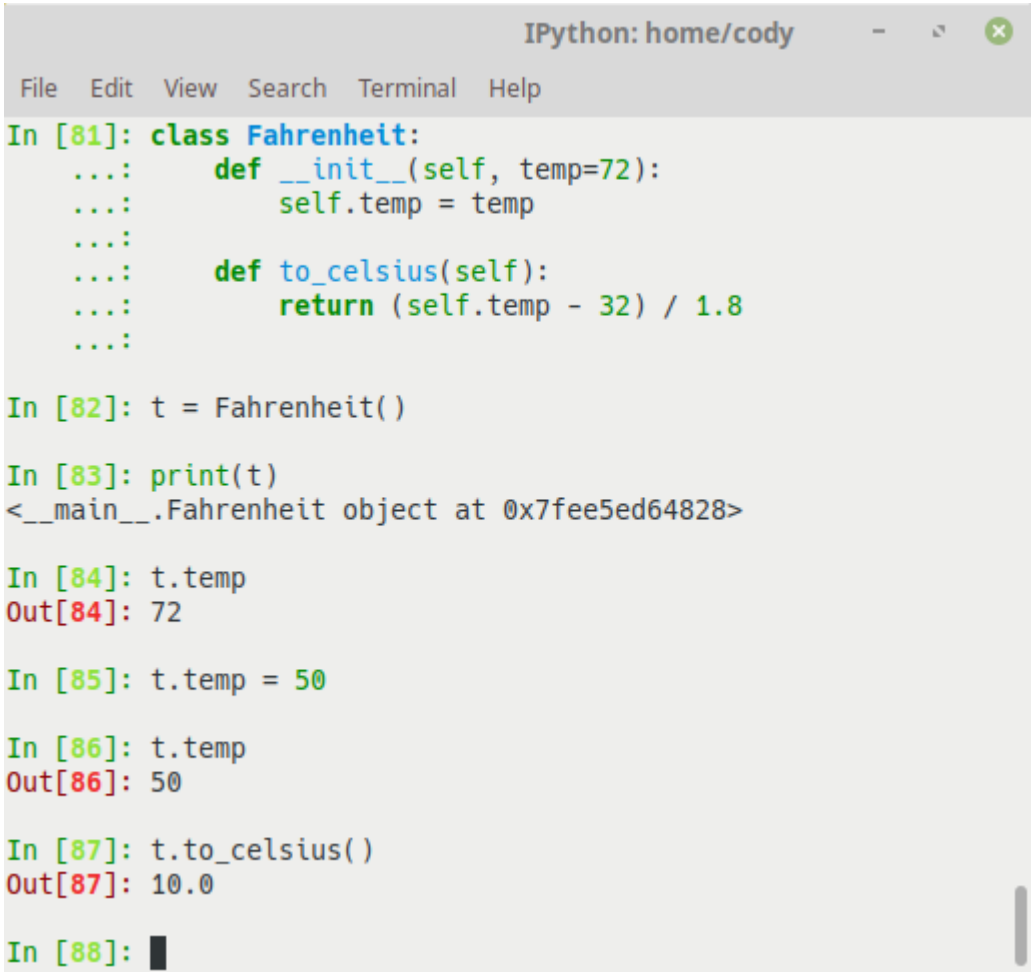
One final thing to mention about operator overloading is that you can make your custom methods do whatever you want. However, common practice is to follow the structure of the built-in methods. That is, if a built-in method creates a new object when called, your overriding method should too. This reduces confusion when other people are using your code. Regarding the preceding example, the built-in method for resolving `*` expressions creates a new object (just as the `+` method does); therefore, the overriding method we created should probably create a new object too, rather than changing the value in-place as it currently does. You're not obligated to "follow the rules" but it does make life easier when things work as expected.

Properties and class and static methods

Properties, class methods, and static methods are additional tools available to OOP development. There are others you will find as you gain Python proficiency, but these are some of the key ones to be aware of, primarily because we will use them in the second half of this book when we write the full-featured application.

Properties

Properties are found in other languages as *getters* and *setters.* As those terms indicate, they are used to get data and set (modify) data. To gain an understanding of getters, setters, and Python properties, we'll start with an example problem:



Temperature class

Here, we create a class that defines the temperature in degrees Fahrenheit. The `__init__()` method takes `temp` as an argument; it also provides a default value for `temp` if none is provided. A conversion method to Celsius is also provided; it simply returns the conversion value but doesn't attempt to assign any values to any variables.

A new instance is created on line 82 and it is verified that it exists on line 83. Line 84 returns the temperature value; because no value was provided during the instance creation, the default value is used.

Line 85 assigns a new value to the `temp` variable, as shown on line 86. Line 87 shows that the `Celsius` conversion method works as expected.

Getters and setters

If we assume that people are using this script to do work, but we want to improve upon it by adding new features, such as preventing a temperature less than absolute zero, we need to make it easy for our users to deal with the change.

This is where getter and setter methods come in. Normally, the direct access to *temp* would be hidden behind these methods, so that the only interface to interact with *temp* would require a method call, as shown in following screenshot:

```
File  Edit  View  Search  Terminal  Help
In [90]: class Fahrenheit:
    ...:     def __init__(self, temp=72):
    ...:         self.set_temp(temp)
    ...:
    ...:     def to_celsius(self):
    ...:         return (self.get_temp() - 32) / 1.8
    ...:
    ...:     def set_temp(self, temp):
    ...:         if temp < -459.67:
    ...:             raise ValueError("Temperature cannot be less than absolute
    ...: zero")
    ...:         self._temp = temp
    ...:
    ...:     def get_temp(self):
    ...:         return self._temp
    ...:

In [91]: f = Fahrenheit()

In [92]: f.get_temp()
Out[92]: 72

In [93]: f.set_temp(-460)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-93-a748fd671613> in <module>()
----> 1 f.set_temp(-460)

<ipython-input-90-6dd57dfdeee7> in set_temp(self, temp)
      8     def set_temp(self, temp):
      9         if temp < -459.67:
---> 10             raise ValueError("Temperature cannot be less than absolute z
ero")
     11         self._temp = temp
     12

ValueError: Temperature cannot be less than absolute zero

In [94]: f.set_temp(32)

In [95]: f.get_temp()
Out[95]: 32

In [96]: f.temp
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-96-31d3e6098624> in <module>()
----> 1 f.temp

AttributeError: 'Fahrenheit' object has no attribute 'temp'

In [97]: f._temp
Out[97]: 32

In [98]:
```

Getter and setter methods

On line 90, we have rewritten the `Fahrenheit` class to include a getter and setter; these are the only ways to interact with the `temp` variable. In addition, we have renamed it to `_temp`; the leading underscore tells Python to treat it as a private variable that should only be accessed by this class. (This isn't completely true, but the reasons why Python doesn't have truly private variables is beyond the scope of this book.)

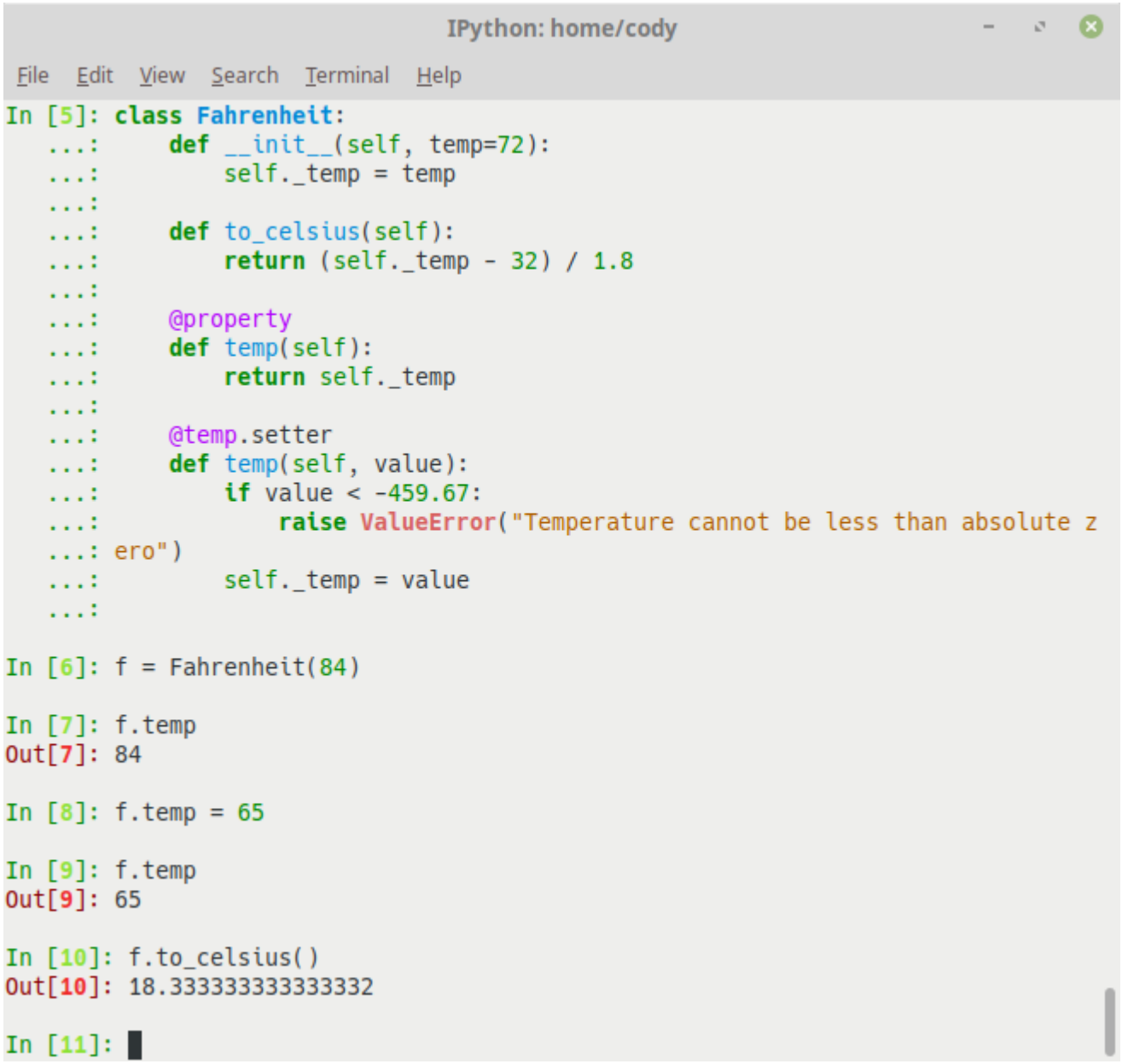
A new instance is created on line 91, and we retrieve the default temperature value on line 92. When we attempt to set the temperature to a value below the absolute value, we receive an error, as desired (line 93).

Lines 94 and 95 set a more realistic temperature and verify it was set, respectively. If we attempt to get the value of `temp` the old-fashioned way (line 96), we receive an error, because that variable is no longer directly accessible that way. If we recognize the private naming convention and provide a leading underscore (line 97), we can still access the temperature directly. However, the purpose of a private variable is to prevent direct access like this.

The problem of using getter and setter methods like this is the issues it causes for users of the program. They now have to change all their programs to use the getter and setter methods, rather than direct variable access. It may be simple with this script, but imagine if we did this with a larger program that had tens or hundreds of getter/setter methods; we have essentially broken any backwards compatibility with our previous versions.

The Python version of getters and setters is `property`. Simply put, a property uses built-in functionality to perform getter/setter methods without requiring the knowledge of specific method names. All changes to the code are internal to the program, so users don't have to worry about the implementation.

The following screenshot demonstrates using the Python property attribute:



Using Python @property

The previously used class is slightly modified on line *5*. Note that the `__init__()` method changed from calling a `set_temp()` method to simply assigning the `temp` argument to the `self.temp`variable. Also, the `to_celsius()` method uses the variable directly, rather than calling the `get_temp()`method.

The `@property` decorator is a Python built-in function that tells Python that the following method defines the getter functionality for a variable. In this case, all we do is return the value of `self.temp` when the variable is referenced elsewhere.

Decorators in Python are any callable objects that can modify a function or method. They allow some additional functionality similar to other languages, such as declaring a method as a class or static method. Decorators allow you to wrap a function or method in another function that can add new functionality, modify arguments, or results, and so on. You write decorators one line above the function definition, beginning with an "at" sign (`@`).

The`@temp.setter` decorator works with the `@property` decorator to define the setter functionality for the indicated variable. Functionality of the following method is no different than the original setter method; it's just wrapped within the decorator so Python knows which variable it applies to.

We make a new instance on line 6, and confirm that the temperature argument was correctly assigned on line 7. Note that no special methods were called, nor was the private `name _temp`required to be used. While the private name is used within the class, the property handles the access to it.

Lines 8 and 9 show off the setter and getter functionality inherent with the property. Again, nothing special has to be done in terms of method or variable calls. The new value is assigned like a normal variable assignment, and the value is returned just by asking for it.

Line 10 shows that the `to_celsius()` method works the same as before. It uses the property abilities to get the value of temperature and doesn't require calling a getter method.

When first making a program, it's probably natural to write getter/setter methods. However, as part of the refactoring process, changing those to properties is advisable due to the simplicity they provide when working with variables, as well as making life easier for users of your software.

Class and static methods

There are several types of methods available for classes. Instance methods are what we have been talking about so far. Class and static methods are other ways to create Python methods.

Class methods are called on the class itself, not just a particular instance of the class. Static methods apply to all instances of a class, not just a specific one.

An example of their use is contained in the following screenshot:

Class and static methods (part 1)

A class about dogs is made on line 10. The initialization method takes arguments for the dog's breed and age. Getter methods are written to return the breed and age, but aren't really necessary because the last

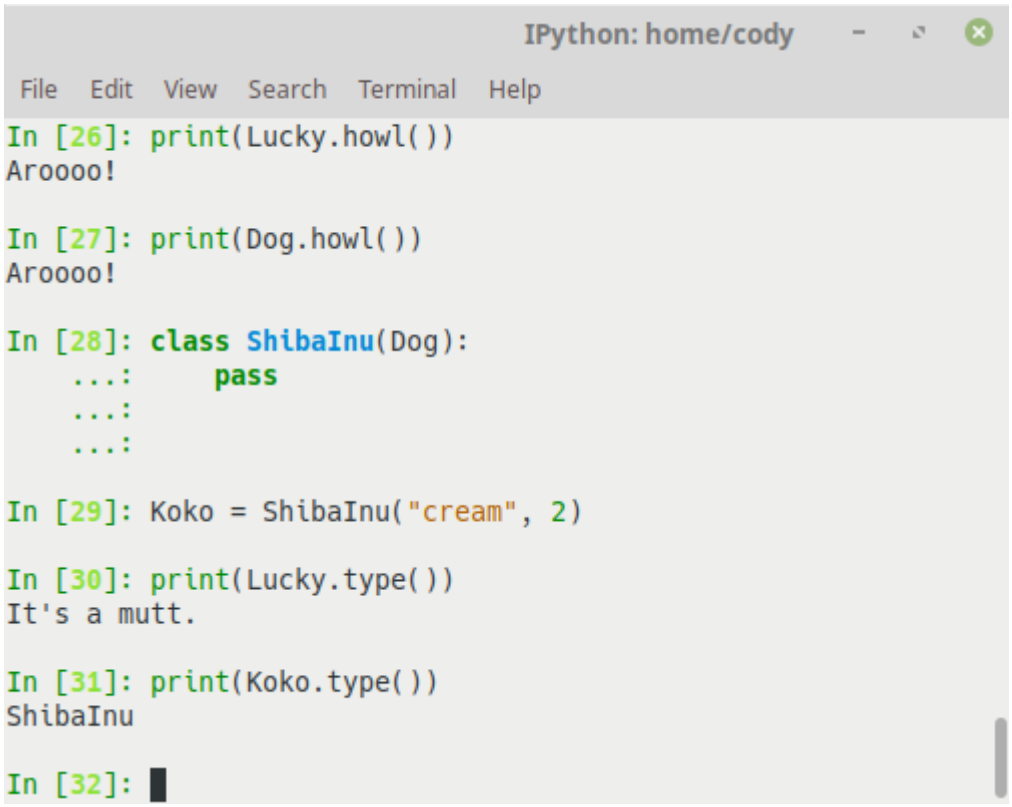
method in the class, `__repr__()`, defines the print formatting used when asking for those values. These getter methods could be changed to Python properties as well.

The first new thing we encounter is the `@staticmethod` decorator. The method defined here can be applied to all instances of the class, as well as the class itself; note that there is no *self* within the parentheses. They are restricted in what data they can access, and they cannot modify the state of an instance or class.

Next, we have `@classmethod`. The method defined here can only be used when calling the class itself; it cannot modify instances. This is because the argument passed in the parentheses is `cls`, rather than `self`. Thus, class methods only know about the class and are oblivious to any instances of the class. For this class method, it checks to see whether the class that called it is `Dog`, or some other class, such as a subclass that inherits this method.

We make an instance of this class on line 11, and then print the breed and age of the dog on lines 12 and 13. If we attempt to do the same with the class itself, the only information we get is that there is an address location for `Dog.breed` but an error is generated when `age` is requested. This is because the class acts as a container for instances, but cannot access normal instance methods.

The program is continued in following screenshot:



Class and static methods (part 2)

Lines 26 and 27 demonstrate accessing the static method. Because there is no `self` or `cls`argument for a static method, it can be used with anything related to the class. In other words, it can be used with any instance of the class, as well as the class itself and any subclasses.

Speaking of subclasses, we make one on line 28. All methods are inherited and we don't need any new ones, so `pass` is used to tell Python to continue. An instance of the subclass is created on line 29.

Lines 30 and 31 demonstrate use of the class method. As stated earlier, when the class method is invoked on the base class `Dog` on line 30, it returns a predefined string. But if called on a subclass (line 31), it returns the name of the subclass.

Summary

In this chapter, we learned about functions, which provide the main operational capability of many programs, as they perform specific tasks while allowing code reuse without copying and pasting. We also learned about object-oriented programming, including what classes and methods are, as well as how to implement them. Finally, we covered different types of OOP methods and how to utilize Python properties to hide the internals of programming logic from users.

In the next chapter, we will look at files and file input/output operations, as well as database creation and access.

Files and Databases

The final built-in object type of Python allows us to access files. Files in Python are different from the previous types I've covered. They aren't numbers, sequences, or mappings; they only export methods for common file processing. Technically, the file is a pre-built C extension that provides a wrapper for the C `stdio.h` (standard input/output) library header. If you already know how to use files in other languages, there isn't much difference in Python.

Files are a way to save data permanently. Apart from a few program listings, nearly everything demonstrated so far is resident only in memory; as soon as you close down Python or turn off your computer, it goes away. You would have to retype everything over if you wanted to use it again.

The files that Python creates are manipulated by the computer's filesystem. Python is able to use operating-system-specific functions to import, save, and modify files. It actually doesn't take much to make cross-platform applications, because the Python **application programming interface** (**API**) handles a lot of the details behind the scenes. All the developer needs to know is which commands to call for a particular operation.

In this chapter, we will cover the following topics:

- File input/output operations
- Python and SQLite databases
- SQLAlchemy and database administration

File I/O

File creation is extremely easy with Python: you simply create the variable that will represent the file, open the file, give it a filename, and tell Python that you want to write to it.

If you don't expressly tell Python that you want to write to a file, it will be opened in read-only mode. This acts as a safety feature to prevent you from accidentally overwriting files. In addition to the standard `w` to indicate writing and `r` for reading, Python supports several other file access modes:

- `a`: Appends all output to the end of the file; it does not overwrite information currently present. If the indicated file does not exist, it is created.
- `r`: Opens a file for input (reading). If the file does not exist, an `IOError` exception is raised.
- `r+`: Opens a file for input and output. If the file does not exist, causes an `IOError` exception.
- `w`: Opens a file for output (writing). If the file exists, it is overwritten. If the file does not exist, one is created.
- `w+`: Opens a file for input and output. If the file exists, it is overwritten; otherwise one is created.
- `ab`, `rb`, `r+b`, `wb`, `w+b`: Opens a file for binary, non-textual input or output. (Note: these modes are supported only on the Windows and macOS platforms. *nix systems don't care about the data type.)

When using standard files, most of the information will be alphanumeric in nature, hence the extra binary-mode file operations. Unless you have a specific need, this will be fine for most of your tasks.

A typical command to open a file to write to might look like this: `open('data.txt', 'w')`. An optional, third argument can be added for buffering control. If you used `open('data.txt', 'w', 0)`, then the data would be immediately written to the file without being held temporarily in memory. This can speed up file operations at the expense of data integrity.

Here is a list of common Python file operations:

- `output = open('/tmp/spam', 'w')`: Create output file (`'w'` means write)
- `input = open('data', 'r')`: Create input file (`'r'` means read, and is the default file operation)
- `append = open('file.txt', 'a')`: Append more data to the end of the file without overwriting
- `S = input.read()`: Read entire file into a single string
- `S = input.read(n)`: Read `n` number of bytes
- `S = input.readline()`: Read next line (through end-line marker)
- `L = input.readlines()`: Read entire file into list of line strings; note that this is different from `read()` in that `readlines()` splits the file into separate lines, placed into a list
- `output.write(S)`: Write string `S` onto file
- `output.writelines(L)`: Write all line strings in list `L` onto file
- `output.close()`: Manual close

Python has a built-in garbage collector, so you don't really need to manually close your files; once an object is no longer referenced within memory, the object's memory space is automatically reclaimed. This applies to all objects in Python, including files.

However, it's recommended to manually close files in large systems; it won't hurt anything and it's good to get into the habit in case you ever have to work in a language that doesn't have garbage collection. In addition, Python for other platforms, such as Jython or IronPython, may require you to manually close files to immediately free up resources, rather than waiting for garbage collection. Also, there are times when system operations have problems and a file is left open accidentally, resulting in a potential memory leak.

The location of the file you are working with can be indicated as either an absolute path (a specific location on a drive) or a relative path (the file location in relation to the current directory); if no path is provided, the current directory is assumed.

Files and streams

Coming from a *nix background, Python treats a file as a data stream: each file is read and stored as a sequential flow of bytes. Each file has an **end-of-file** (**EOF**) marker denoting when the last byte of data has been read from it. Frequently, programs will read a file in pieces rather than loading the entire file into memory at one time. When the end-of-file marker is reached, the program knows there is nothing further to read and can continue with whatever processing it needs to do.

When a file is read, such as with a `readline()` method, the end of the file is shown at the command line with an empty string; empty lines are just strings with an end-of-line character. The following screenshot shows how this looks:

IPython: home/cody
File Edit View Search Terminal Help
In [22]: this_file = open("myfile.txt", "w")
In [23]: this_file.write("Hello, my little text file.")
Out[23]: 27
In [24]: this_file.close()
In [25]: that_file = open("myfile.txt")
In [26]: that_file.readline()
Out[26]: 'Hello, my little text file.'
In [27]: that_file.readline()
Out[27]: ''

EOF example

A file object is opened in line 22, with the `w` flag provided to indicate that the file is to be written to; the default mode when opening a file is read-only, as seen in line 25.

Line 23 writes a short text string to the file. When completed, Python 3 tells us how many characters were written to the file. Python 2.x doesn't automatically provide this feedback.

The file is manually closed in line 24, then reopened as a new filename in line 25. Because the default mode is read-only, we didn't have to provide the `r` flag.

The file is read in line 26, accepting data until the end of the line. The line read in is then printed to the screen. When we attempt to read the next line in the file (line 27), an empty string is returned to tell us there is nothing left.

If you want to put a new line in the file you're writing to (through the interactive shell), you must include the newline (`\n`) character every time you want to denote the end of a line; Python's `write()` method doesn't include it automatically. This is demonstrated in the following screenshot:

IPython: home/cody
File Edit View Search Terminal Help
In [28]: new_file = open("myfile.txt", "a")
In [29]: new_file.write("Here's a new line.")
Out[29]: 18
In [30]: new_file.write("Here's another line.")
Out[30]: 20
In [31]: new_file.close()
In [32]: read_file = open("myfile.txt")
In [33]: read_file.readline()
Out[33]: "Hello, my little text file.Here's a new line.Here's another line."
In [34]: newer_file = open("yourfile.txt", "w")
In [35]: newer_file.write("This is a new file.\n")
Out[35]: 20
In [36]: newer_file.write("See how the lines are separate?\n")
Out[36]: 32
In [37]: newer_file.close()
In [38]: read_file = open("yourfile.txt")
In [39]: read_file.readlines()
Out[39]: ['This is a new file.\n', 'See how the lines are separate?\n']
In [40]: print(read_file.readlines())
[]
In [41]:
Writing and reading files

With line 28, we reopen the file using `append` mode; this allows us to write to the file without overwriting the data that is already present. Lines 29 and 30 add two additional lines to the file, then it is closed in line 31.

If we open the file to read it (lines 32 and 33), we see that all the lines added to the file are put into one string; there is no default separation between them, even though they were inserted separately.

So, we make a brand-new file in line 34. With lines 35 and 36, we write lines to the file but ensure that the newline character `/n` is added at the end of each line. The file is then closed and reopened.

If we send the data in the file to the `readlines()` method, we see a list of the two strings, both showing the newline characters. If we try to print more (line 40), we are presented with an empty string, as once the file is processed by one of the read methods, the data is purged from the variable.

If you want to view the file information as it would normally be seen, that is, outside of a list, you'll have to use the regular `read()` method, as shown in the following screenshot:

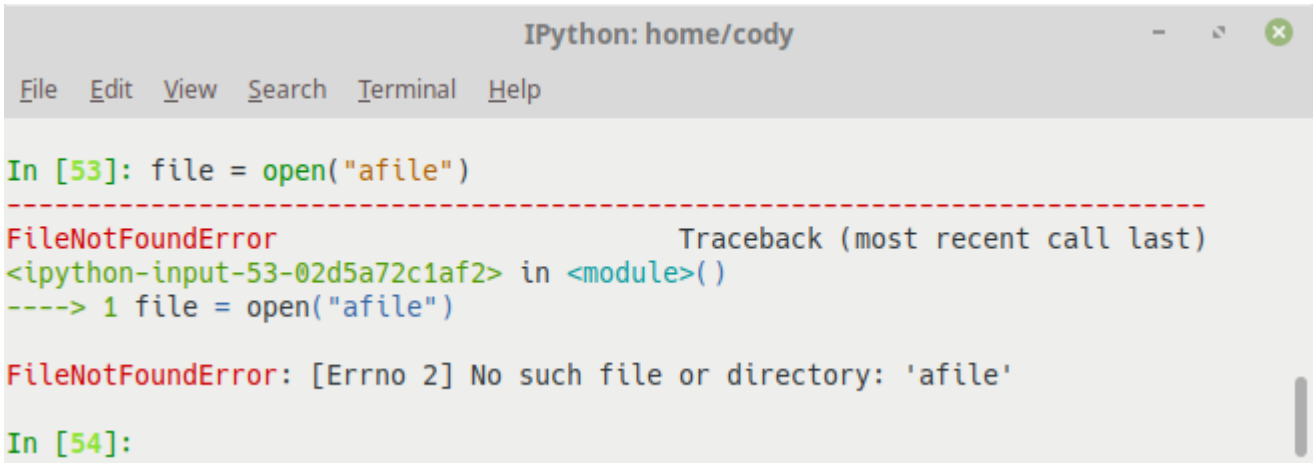
File read() method

Using the `read()` method reads the entire file contents, at once, into a string. The `print()` function then allows you to print a reader-friendly version of the file, one that doesn't show the newline characters.

If you want to save the buffered data to the file without closing out the file, you can use the `flush()` method instead of `close()`. As the name implies, the buffer is flushed out, sending the stored data to the actual file location while leaving the buffer open for more data. Thus, you can continue working with data without continuously opening and closing the file. For example, instead of using `new_file.close()` in line 31 in the previous screenshot, we could have used `new_file.flush()` to immediately write all data to the file, but `new_file` would remain open for more writing.

Reading from a file

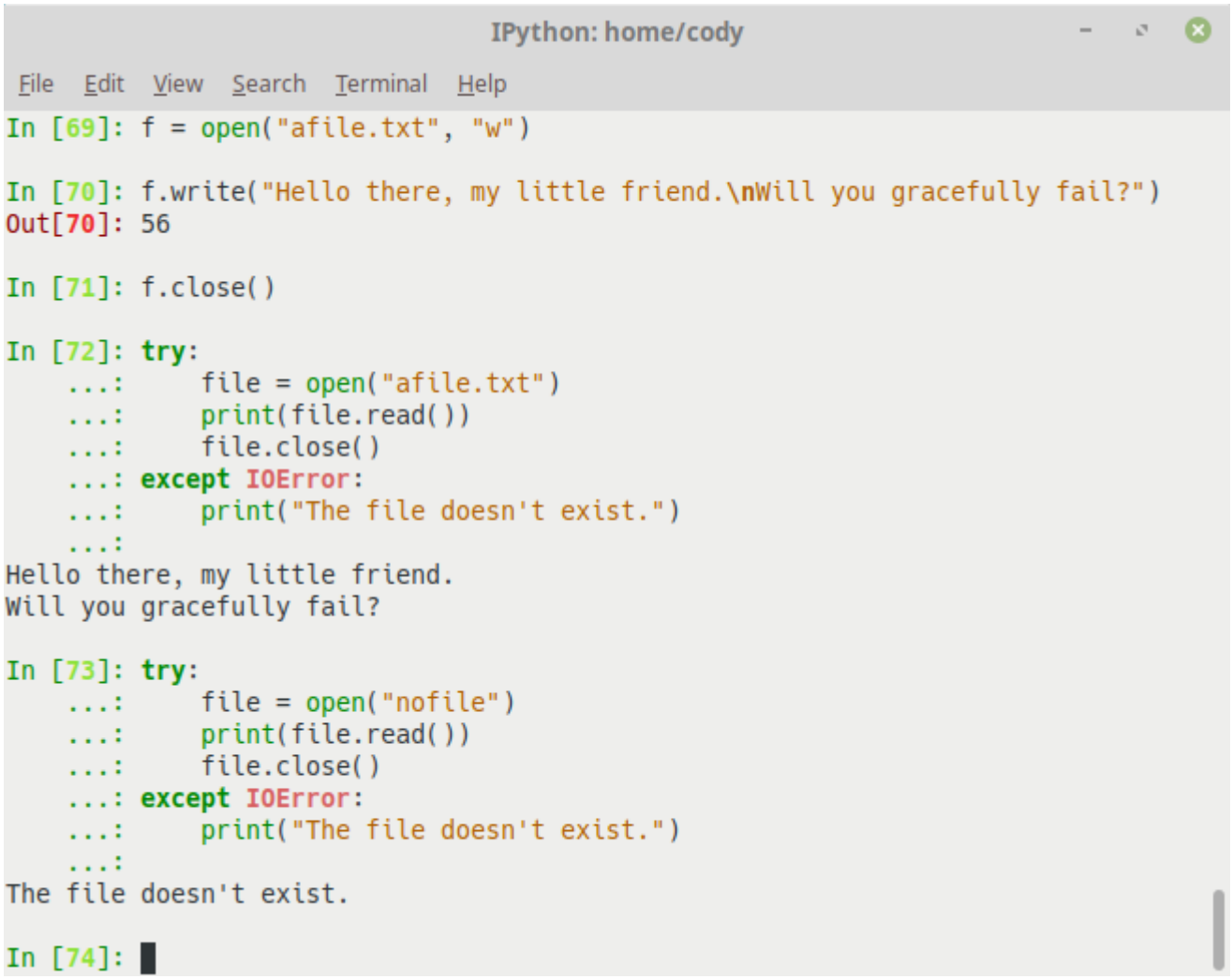
Note that in the list of file operations in the *File I/O* section, the standard read modes produce an **input/output** (**I/O**) error if the file doesn't exist. If you end up with this error, your program will halt and give you an error message, like in the following screenshot:



File IOError

To fix this, you should always open files in such a way as to catch the error before the program crashes. When you are performing an operation where there is a potential for an exception to occur, you should wrap that operation within a `try`/`except` code block. This will attempt to run the operation; if an exception is thrown, you can catch it and deal with it gracefully. Otherwise, your program will error out, potentially causing problems for the user.

Exception handling with files is demonstrated in the following screenshot:



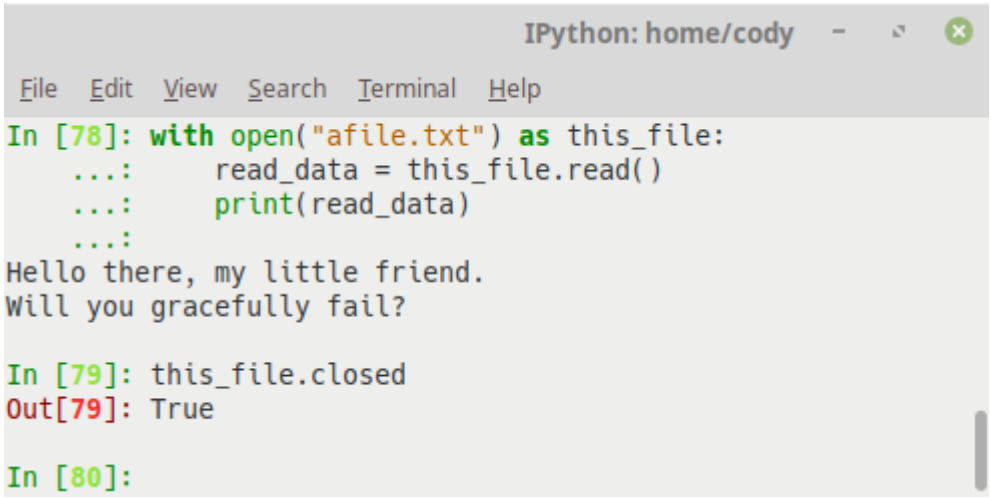
Catching file exceptions

A file is created for writing to in line 69, then we add a string to it and close it (lines 70 and 71, respectively).

A `try`/`except` block is written to attempt opening and reading the file in line 72. Because the file exists, it runs as expected and prints the data in the file.

In line 73, we make a new `try`/`except` block, this time with a file that doesn't exist. Because it doesn't exist, it would normally print an error message like the previous screenshot. With the exception-catching code, we look explicitly for the `IOError`, and when it is generated, we print a nice message to the user. Alternatively, we could add code that would provide other functionality, such as automatically looking in a different directory, or presenting the user the option to indicate where the file is located.

One way to open and then automatically close a file, so you don't have to manually close it or wait for the garbage collector, is to use the `with` keyword when working with file objects. This ensures the file is automatically closed after it is no longer needed, even if an exception arose during processing. The following screenshot provides an example of this:

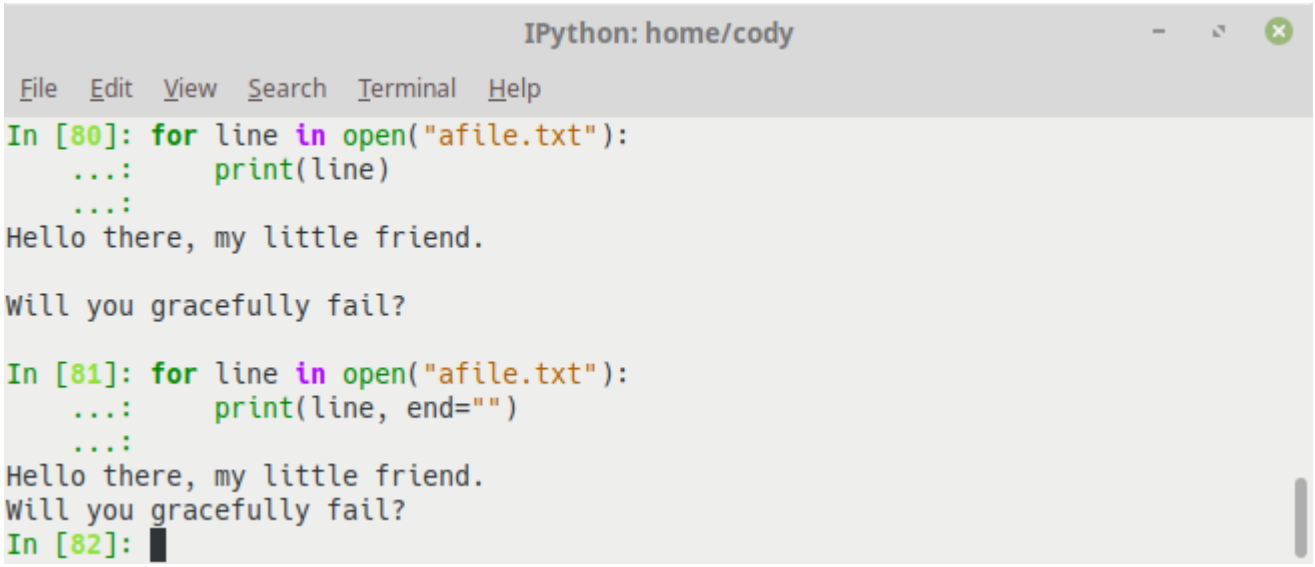


Auto-closing files

Rather than opening a file like we have in previous examples, line 78 shows how to use the `with` keyword to open a file and assign it to a variable name. We can process the file like normal, in this case simply printing the file data as before. When we check to see if the file has been closed in line 79, Python tells us that the file is, indeed, closed.

Iterating through files

Iteration is used frequently with files; iteration can be used to read the information in the file and process it in an orderly manner. It also limits the amount of memory taken up when a file is read, which not only reduces system resource use but can also improve performance. The following screenshot demonstrates this:



Simple file iteration

Basically, the code in line 80 opens a temporary file in memory and reads each line from `afile.txt`, printing each one to the screen, until the end-of-file marker is reached. Unlike previous examples, using a `for` loop won't print the EOF marker because, once that marker is reached, file processing is aborted.

Line 81 shows the same thing, except that the `print()` function is provided with an argument for how to end each line. In this case, we just want to have an empty value to cause the lines to be printed without an intervening space.

When iterating through files, it is important to note that the `readlines()` method requires the file to be placed in memory before it can be processed; for large files, this can result in a performance hit or out-of-memory errors. Therefore, it may be better to look at one of the other file methods to process each line individually or come up with another solution.

Seeking

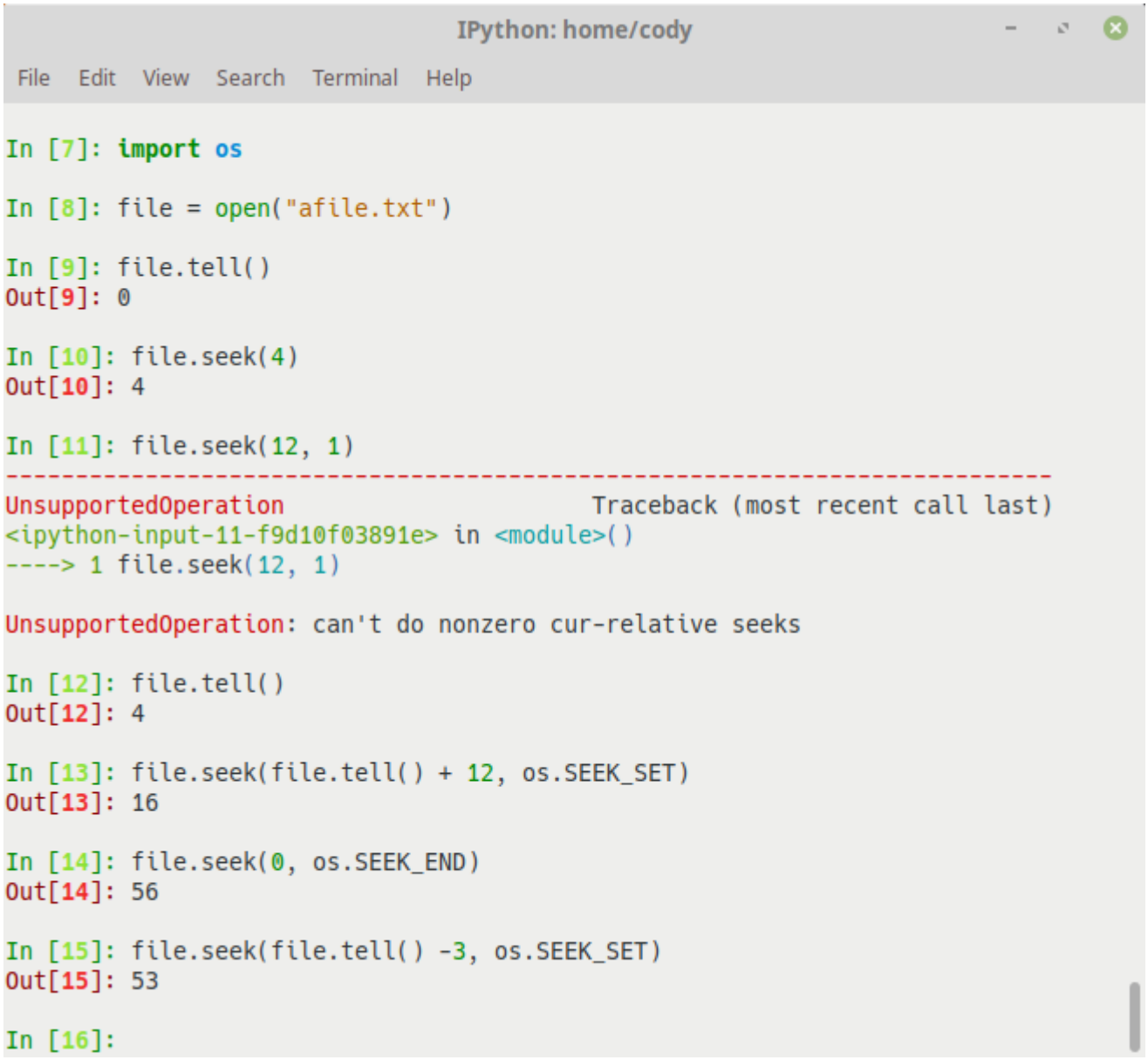
Seeking is the process of moving a pointer within a file to an arbitrary position. This allows you to get data from anywhere within the file without having to start at the beginning every time.

The `seek()` method can take several arguments. The first argument (`offset`) is the starting position of the pointer. The second, optional, argument is the seek direction from where the offset starts. The default value is `0` which indicates an offset relative to the beginning of the file, `1` is relative to the current position within the file, and `2` is relative to the end of the file.

The `tell()` method returns the current position of the pointer within the file. This can be useful for troubleshooting (to make sure the pointer is actually in the location you think it is) or as a returned value for a function.

One caveat to this is that text files can only be sought relative to the beginning of the file (starting with Python 3.2). Binary files don't suffer from this limitation.

As a best practice, it is advised that, instead of using `0`, `1`, or `2` for the offset starting position, you should import the `os` library and use the values `os.SEEK_SET`, `os.SEEK_CUR`, and `os.SEEK_END`. This is because the `os` library will use whatever values the operating system uses for seeking, rather than hardcoded numbers, just in case the OS does something different. This also ties in to the `tell()` method, as it can be used to help seek within a text file, as demonstrated in the following screenshot:



File seeking

The `os` module is imported in line 7 and the file is opened. Line 9 returns the current location of our position within the file; zero indicates the beginning of the file.

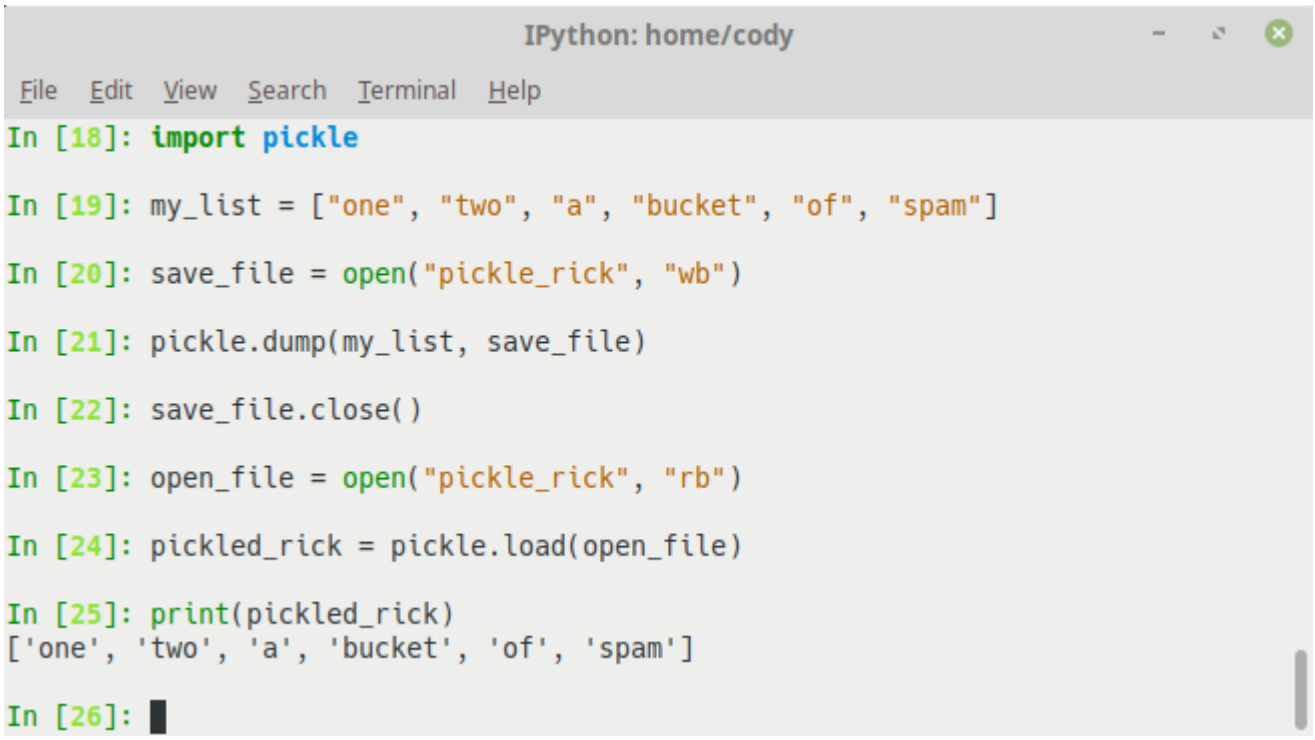
In line 10, we move the pointer ahead in the file by four characters. But when we try to move ahead another 12 characters in line 11, we receive an error. The error indicates that Python can't perform relative seeking from the current location; it can only do this from the beginning (zero) position. To fix this, we first confirm our position (line 12), then use `tell()` and `os.SEEK_SET` in line 13 to move ahead 12 characters.

Serialization

Since files work with text strings by default, to save Python data types like dictionaries or tuples, you could convert them to strings first, but an easier method is to serialize the data.

Serialization (pickling) allows you to save non-textual or binary information or transmit it over a network. Pickling essentially takes any data object, such as dictionaries, lists, or even class instances, and converts it into a byte set that can be used to reconstitute the original data. When pickling data, you must use the binary mode for files, as the data is being stored in the raw, rather than the normal text strings file operations are used to.

The following screenshot shows how to use the `pickle` library:


IPython: home/cody
File Edit View Search Terminal Help
In [18]: import pickle
In [19]: my_list = ["one", "two", "a", "bucket", "of", "spam"]
In [20]: save_file = open("pickle_rick", "wb")
In [21]: pickle.dump(my_list, save_file)
In [22]: save_file.close()
In [23]: open_file = open("pickle_rick", "rb")
In [24]: pickled_rick = pickle.load(open_file)
In [25]: print(pickled_rick)
['one', 'two', 'a', 'bucket', 'of', 'spam']
In [26]:
List pickling

We first import the `pickle` library in line 18, create a list of strings in line 19, and open a save file in line 20.

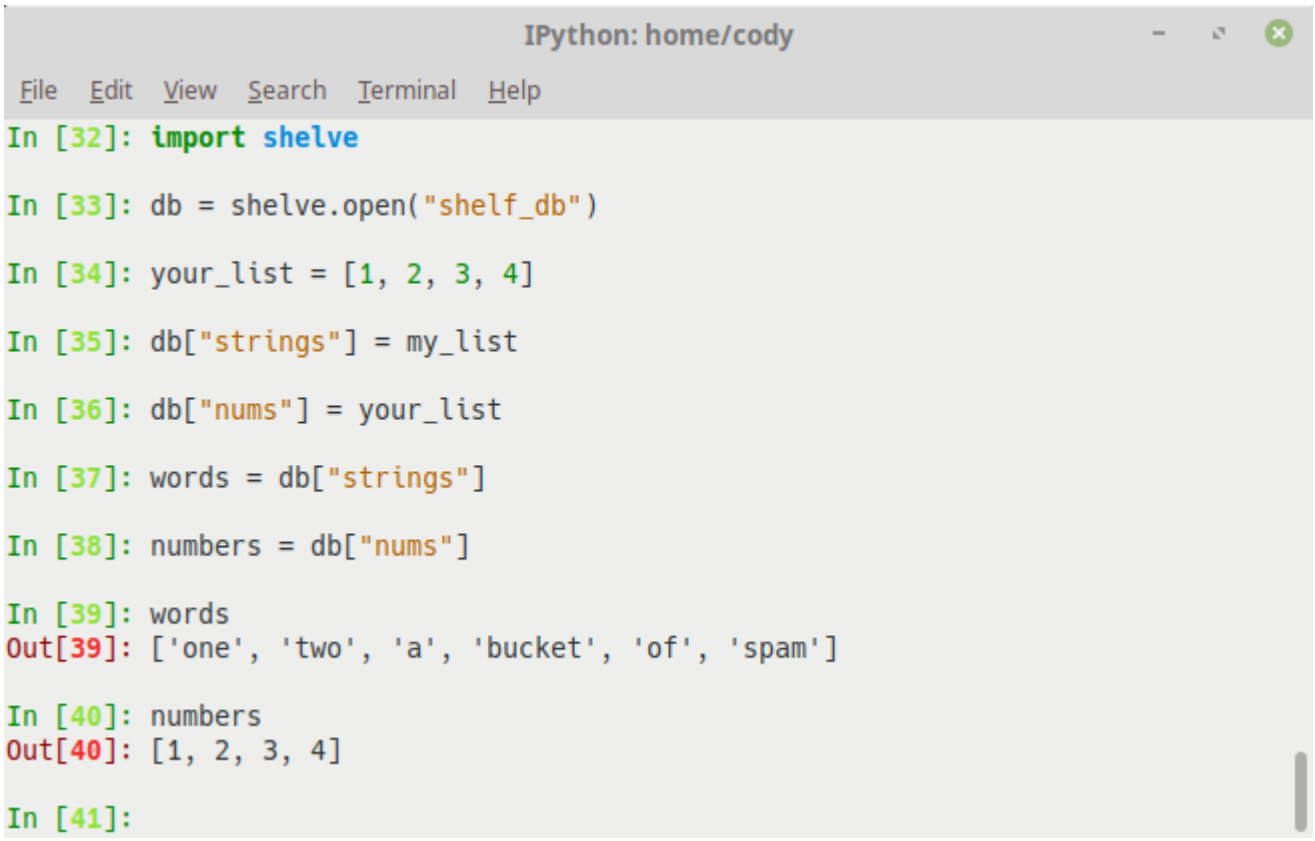
In line 21, the list is pickled to the file, then the file is closed. If you were to look at the current directory now, you would see that the pickled file has been created.

The file is reopened in line 23 and the pickled data loaded into a variable (line 24). When we print the new variable, we see the original list is returned.

In Python 3, only the `pickle` library is available. For Python 2.x, there are two different `pickle`libraries for Python: `cPickle` and `pickle`. Since Python is interpreted, it runs a bit slower compared to compiled languages, such as C. Because of this, Python has a precompiled version of `pickle` that was written in C; hence `cPickle`. Using `cPickle` makes your program run faster; if you decide to use it, you might consider importing `cPickle` as `pickle`, so you're not always typing `cPickle`.

Shelves are similar to pickles except that they pickle objects to an access-by-key database, much like dictionaries; as a matter of fact, the `pickle` library provides the backend functionality. Shelves allow you to simulate a random-access file or a database. It's not a true database but it often works well enough for development and testing purposes. Alternatively, shelves can be used when the values will be class instances, recursive data types, containers with multiple sub-objects, and similar Python objects.

The following screenshot shows how the `shelve` library is used:


IPython: home/cody
File Edit View Search Terminal Help
In [32]: import shelve
In [33]: db = shelve.open("shelf_db")
In [34]: your_list = [1, 2, 3, 4]
In [35]: db["strings"] = my_list
In [36]: db["nums"] = your_list
In [37]: words = db["strings"]
In [38]: numbers = db["nums"]
In [39]: words
Out[39]: ['one', 'two', 'a', 'bucket', 'of', 'spam']
In [40]: numbers
Out[40]: [1, 2, 3, 4]
In [41]:
Shelve object persistence

The `shelve` library is imported, and a new shelve object is created (lines 32 and 33). A new list is created in line 34, then the shelve object is populated (lines 35 and 36) with the list of strings from the preceding screenshot, as well as the list from line 34.

The two lists within the `shelve` object are assigned to new variables (lines 37 and 38). We can verify this action by looking at the individual variables in lines 39 and 40.

One thing to note about pickling: there is no data security/integrity performed. Passing pickled objects around or storing them doesn't prevent the data from becoming corrupted or being maliciously modified. Never unpickle data from untrusted or unauthenticated sources.

Python and SQLite

Databases are popular for many applications, especially for use with web applications or customer-oriented programs. Databases are good when discrete structures are to be operated on, such as a customer list that has phone numbers, addresses, past orders, and so on. A database can store a lump of data and allow the user or developer to pull the necessary information, without regard to how the data is stored. Additionally, databases can be used to retrieve data randomly, rather than sequentially. For pure sequential processing, a standard file is better.

A **database** (**DB**) is simply a collection of data, placed into an arbitrary structured format. The most common DB is a relational database; tables are used to store the data and relationships can be defined between different tables. **Structured Query Language** (**SQL**) is the language used to work with most DBs. (SQL can either be pronounced as discrete letters **S**-**Q**-**L** or as a word, sequel.)

SQL provides the commands to query a database and retrieve or manipulate information. The format of a query is one of the most powerful forces when working with DBs; an improper query won't return the desired information, or worse, it will return the wrong information. SQL is also used to input information into a DB.

While you can interact directly with a DB using SQL, as a programmer you have the liberty of using Python to control much of the interactions. You will still have to know SQL so you can populate and interact with the DB, but most of the calls to the DB will be with the Python **database application programming interface** (**DB**-**API**).

Starting with v2.5, Python has included SQLite, a lightweight SQL-based database management system. SQLite is written in C, so it's quick.

It also creates the database in a single file, which makes implementing a DB fairly simple; you don't have to worry about all the issues of having a DB spread across a server. However, it does mean that SQLite is better suited to either development purposes or small, standalone applications. If you are planning on using your Python program for large-scale systems, you'll want to move to a more robust database, such as **PostgreSQL** or **MySQL**.

Working with databases

This book is not intended to be a database or SQL primer, but it will help to have an understanding of how traditional interaction with a database using SQL occurs.

First, consider a database to be one or more tables, just like a spreadsheet. The vertical columns comprise different fields or categories; they are analogous to the fields you fill out in a form. The horizontal rows are individual records; each row is one complete record entry. Here is a pictorial summary, representing a customer list. The table's name is **Customers**:

Index	LName	FName	Address	City	State
0	Johnson	Jack	123 Easy St.	Anywhere	CA
1	Smith	John	312 Hard St.	Somewhere	NY

The only column that needs special explanation is the **Index field**. This field isn't required but is highly recommended. You can name it anything you want but the purpose is the same. It is a field that provides a unique value to every record; it's often called the **primary key**field. The primary key is a special object for most databases; simply identifying which field is the primary key will automatically increment that field as new entries are made, thereby ensuring a unique data object for easy identification. The other fields are simply created based on the information that you want to include in the database.

To make a true relational database, one table needs to refer to one or more different tables in some fashion.

If you wanted to make an order-entry database, you could make another table that tracks an order and relate that order to the preceding customer list, like so:

Key	Item_title	Price	Order_Number	Customer_ID
0	Boots	55.50	4455	0
1	Shirt	16.00	4455	0
2	Pants	33.00	7690	0
3	Shoes	23.99	3490	1
4	Shoes	65.00	5512	1

This table is called **Orders**. This table shows the various orders made by each person in the customer table. Each entry has a unique key and is related to **Customers** by the **Customer_ID** field, which is the Index value for each customer.

The code listing next shows how this database could be created:

```
# sqlite_db.py (part 1)
# import sqlite3
1
2 connection = sqlite3.connect("Customers.db") # The .db extension is optional
3 cursor = connection.cursor() # Executes SQL queries
4
5 # Alternative DB created only in memory
6 # mem_conn = sqlite3.connect(":memory:")
7 # cursor = mem_conn.cursor()
8
9 # Create the table to hold entries
10 cursor.execute("""
11     CREATE TABLE Customers
12     (id INTEGER PRIMARY KEY,
13     LName TEXT,
```

Here, we import the `sqlite3` database module in line 1, create the database in line 3, and establish a connection to the DB in line 4.

Lines 6-8 demonstrate how to create a database strictly within memory and not writing the data to disk. This is fine for testing, but shouldn't be used in production because, if the power fails on the computer, the data in memory is lost.

Starting with line 11, we create the DB table that will hold our customer information:

```
# sqlite_db.py (part 2)
1    FName TEXT,
2    Address TEXT,
3    City TEXT,
4    State TEXT)
5    """)
6
7 cursor.execute("""
8     CREATE TABLE Orders
9     (id INTEGER PRIMARY KEY,
10     Item_title TEXT,
11    Price FLOAT,
12    Order_Number INTEGER,
13    customer_id INTEGER,
14    FOREIGN KEY (customer_id) REFERENCES Customers(id))
15    """)
```

Here, we complete the table declarations in lines 1-5. Then, in lines 7-15, we create the table to hold the customer orders:

```
# sqlite_db.py (part 3)
1 def customer_insert(last_name, first_name, address, city, state):
2     sql = "INSERT INTO Customers VALUES (?, ?, ?, ?, ?, ?)"
3     cursor.execute(sql, (None, last_name, first_name, address, city, state))
4     return cursor.lastrowid # Get ID of object
5
6 def order_insert(item, price, order_num, customer_id):
7     sql = "INSERT INTO Orders VALUES (?, ?, ?, ?, ?)"
```

```
8     cursor.execute(sql, (None, item, price, order_num, customer_id))
9     return cursor.lastrowid
10
11     johnson_id = customer_insert("Johnson", "Jack", "123 Easy St.", "Anywhere",
"CA")
12     johnson_order1 = order_insert("Boots", 55.50, 4455, johnson_id)
13     johnson_order2 = order_insert("Shirt", 16.00, 4455, johnson_id)
14     johnson_order3 = order_insert("Pants", 33.00, 7690, johnson_id)
```

Here, lines 1-4 create a function that populates the customer table with supplied information, while lines 6-9 do the same thing for the order table.

Lines 11-14 provide the data that will be used to fill the customer and order table for the first customer:

```
# sqlite_db.py (part 4)
1 smith_id = customer_insert("Smith", "John", "312 Hard St.",           "Somewhere",
"NY")
2 smith_order1 = order_insert("Shoes", 23.99, 3490, smith_id)
3 smith_order2 = order_insert("Shoes", 65.00, 5512, smith_id)
4
5 connection.commit() # Write data to database
6 cursor.close() # Close database
```

Here, we populate the tables with information about the second customer and orders in lines 1-3. Lines 5 and 6 put the data into the DB and then close the connection to the DB.

Using SQL to query a database

To query a table using SQL, you simply tell the database what it is you're trying to do. If you want to get a list of the customers or a list of orders in the system, just select what parts of the table you want to get.

Note that the following code snippets are not Python-specific. Additionally, SQL statements are not case-sensitive but are usually written in uppercase for clarity, so you know what words are SQL-related.

Returning data with SQL is shown in following snippet:

```
SELECT * FROM Customers
```

The command simply pulls everything from **Customers_table** and prints it. The printed results may be textual or have grid lines, depending on the environment you are using, but the information will all be there.

You can also limit the selection to specific fields, such as following example:

```
SELECT Last_name, First_name FROM Customers
SELECT Address FROM Customers WHERE State == "NY"
```

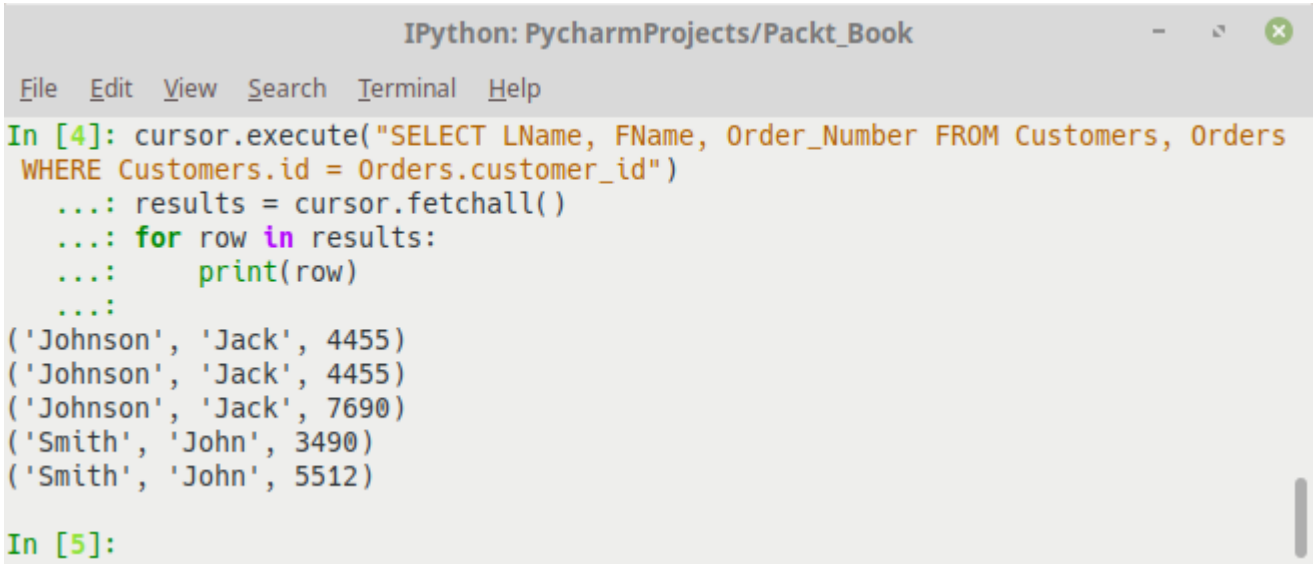
The second SQL query uses the `WHERE` statement, which returns a limited set of information based on the condition specified. In this case, only the addresses of customers who live in New York state would be returned.

Limiting a query in this manner is a good idea because it limits the results you have to process and it reduces the amount of memory being used. Many system slowdowns can be traced to bad DB queries that return too much information and consume too many resources.

To combine the information from two tables, that is, to harness the power of relational databases, you have to join the tables in the query, as demonstrated here:

```
SELECT Last_name, First_name, Order_Number FROM Customers, Orders WHERE Customers.id =
Orders.customer_id
```

This should give you something that looks like the following screenshot:



SQL query results

Again, the formatting may be different depending on the system you are working with, but it's the information that counts.

Creating a SQLite database

To use the SQLite database, you simply import it like any other library. Once imported, you have to make a connection to it; this creates the database file. A cursor is the object within SQLite that performs most of the functions you will be doing with the DB.

The following code listing demonstrates the creation of a SQLite database:

```
# tools_db.py (part 1)
1 import sqlite3
2
3 connection = sqlite3.connect("Tools.db") # The .db extension  is optional
4 cursor = connection.cursor() # Executes SQL queries
5
6 # Create the table to hold entries
7 cursor.execute("""
8     CREATE TABLE Tools
9     (id INTEGER PRIMARY KEY,
10     name TEXT,
11     size TEXT,
12     price INTEGER)
13     """)
```

The `sqlite3` library is imported in line 1. Lines 2 and 3 create the connection to the DB file and the `cursor` object, respectively.

Lines 6-13 create the table that will hold all the DB entries. In this case, the name of the table is `Tools`, a primary key is provided to ensure each entry has a unique identifier, two text entries are created, and so is a numeric entry:

```
# tools_db.py (part 2)
1 # Populate table
2 for item in (
3     (None, "Box Knife", "Small", 15),
4     (None, "Drill", "Medium", 35),
5     (None, "Axe", "Large", 55),
6     (None, "Putty Knife", "Small", 25),
7     (None, "Hammer", "Small", 25),
8     (None, "Screwdriver", "Small", 10),
9     (None, "Crowbar", "Large", 60),
10  ):
11      cursor.execute("INSERT INTO Tools VALUES (?, ?, ?, ?)", item)
12
13 connection.commit() # Write data to database
14 cursor.close() # Close database
```

In part 2, lines 2-11 actually populate the DB with data. The entries are comma-separated to match the entries provided during table creation. The value `None` corresponds to the primary key; we don't have to provide a value as SQLite will automatically increment the key value for each new entry. The rest of the values in each entry apply to the item's name, size, and price.

Line 11 calls `cursor.execute()` to stage the data that will be added to the DB. The question marks are used to prevent a SQL injection attack, where a SQL command is passed to the DB as a legitimate value. The DB would process the command as a normal, legitimate command which could delete data, change

data, or otherwise compromise your DB. The question marks act as a substitution value to prevent this from occurring.

Lines 13 and 14 write the data to the DB then close the connection. The `commit()` command is required to actually put the data into the DB; until this is done, the data is only staged for filling the DB. This allows multiple data staging to occur, with only a single commit being required.

Retrieving data from a database

To pull the data from an SQLite DB, you just use the SQL commands that tell the DB what information you want and how you want it formatted. If the data retrieval is part of the same SQLite program, that is, if the creation of the database and subsequent retrieval are in the same file, then you don't need to include it again; just make sure you haven't closed the cursor connection until the end. Otherwise, you will have to create a connection to the desired DB and recreate the cursor within the new file.

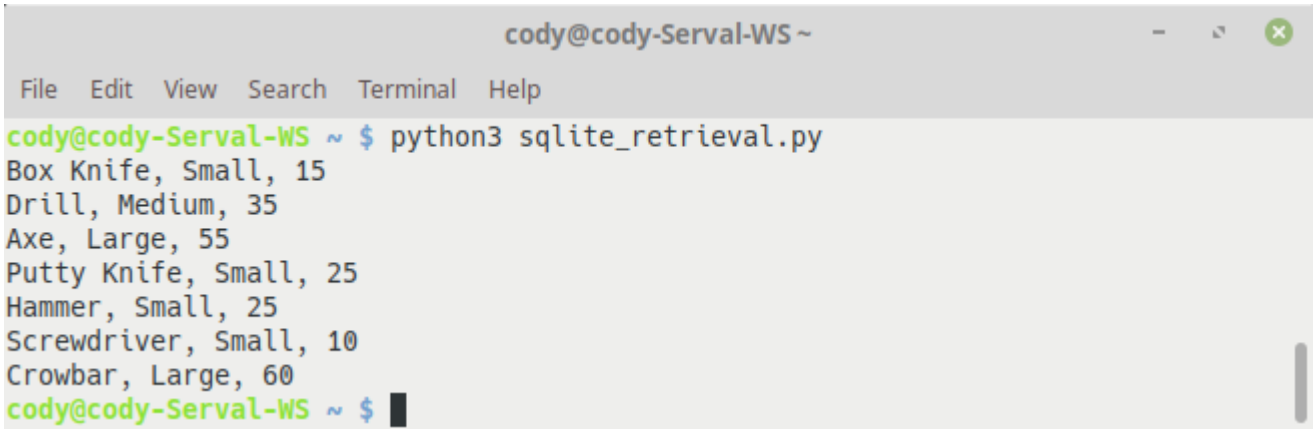
The code listing here modifies the previous code to demonstrate data retrieval:

```
# db_retrieval.py
1 import sqlite3
2
3 connection = sqlite3.connect("Tools.db")
4 cursor = connection.cursor()
5
6 cursor.execute("SELECT name, size, price FROM Tools")
7 toolsTuple = cursor.fetchall()
8 for entry in toolsTuple:
9     name, size, price = entry # Unpack the tuples
10     item = ("{}, {}, {}".format(name, size, price))
11     print(item)
```

Lines 1-4 are only necessary if you are writing a new program; if you want to retrieve the data from within the same program that filled the DB, then you only need to include lines 6-11.

The new commands start with line 6. Here, we execute the command that pulls the desired columns from the `Tools` table. Line 7 tells SQLite to return all entries; you could also specify single entries if desired.

Lines 8-11 is a simple `for` loop that iterates over the entries returned by SQLite and prints the results, which we see in the following screenshot:

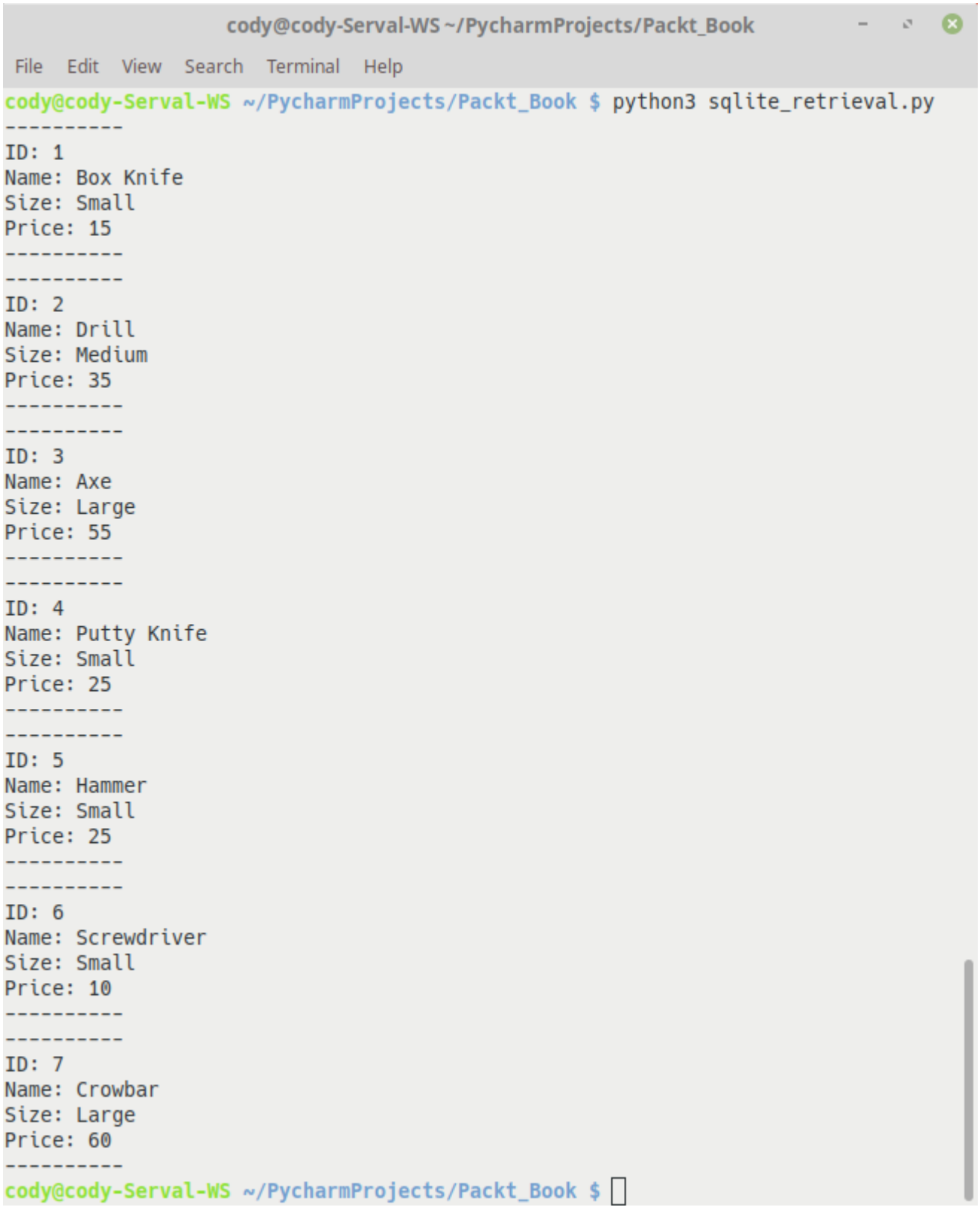


SQLite retrieval results

Alternatively, if you want to print out more structured tables, you can use something like this code:

```
# pretty_print.py
cursor.execute("SELECT * FROM Tools")
for row in cursor:
    print ("-" * 10)
    print ("ID:", row[0])
    print ("Name:", row[1])
    print ("Size:", row[2])
    print ("Price:", row[3])
    print ("-" * 10)
```

When run, the output should look like the following screenshot:



Output of pretty SQL query

Obviously, you can mess around with the formatting to present the information as you desire, such as giving columns with headers, including or removing certain fields, and so on.

SQLite database files

SQLite will try to recreate the database file every time you run the program. If the DB file already exists, you will get an `OperationalError` exception. The easiest way to deal with this is to simply catch the exception and ignore it, as demonstrated here.

Dealing with existing databases:

```
try:
    cursor.execute("CREATE TABLE Foo (id INTEGER PRIMARY KEY, name TEXT)")
    except sqlite3.OperationalError:
    pass
```

This will allow you to run your database program multiple times (such as during creation or testing) without having to delete the DB file after every run.

You can also use a similar `try`/`except` block when testing to see if the DB file already exists; if the file doesn't exist, then you can call the DB creation module. This allows you to put the DB creation code in a separate module from the main program, calling it only when needed.

SQLAlchemy

When working with databases, it can be difficult and time-consuming to deal with writing SQL queries all the time, especially if you don't deal with databases on a regular basis. Luckily, there are many Python libraries available to help with this problem. One of the most popular is **SQAlchemy**.

SQLAlchemy is an **object-relational mapper** (**ORM**), a utility that eliminates the issue of writing raw SQL queries and replaces it with more Python-centric code. ORM tools convert data between incompatible type systems in OOP languages, because these languages frequently use types that are more complex than low-level, primitive types. In other words, an ORM deals with more complex objects, whereas SQL expects primitives and creates relations between them.

Because SQLAlchemy is a third-party library, the easiest way to install it is through `pip`. `pip` is a standard Python package tool, much like `apt` is the package tool for Debian-based Linux. To install SQLAlchemy, use the following command.

Installing software with `pip`:

```
$ pip3 install sqlalchemy
```

This command can be used to install nearly any Python package. The PyPI website can be used to locate particular packages for installation.

Writing a SQLAlchemy database

When working with SQLAlchemy, there are three main components to work with:

- The `table`, which functions as a normal DB table
- The `mapper`, which maps a Python class to a table
- The `class` object, which handles how a DB record maps to a Python object

One beneficial aspect of SQLAlchemy is that all of these items can be placed within the same location, rather than having separate files. The SQLAlchemy `declarative` contains the definitions of these objects.

To convert the previous SQLite database into a SQLAlchemy database, we first create the declarative program, as shown here:

```
# sqlalchemy_declarative.py
1 from sqlalchemy import Column, Integer, String, create_engine
2 from sqlalchemy.ext.declarative import declarative_base
3
4 Base = declarative_base()
5
6 class Tools(Base):
7     __tablename__ = "tools"
8     id = Column(Integer, primary_key=True)
9     name = Column(String(length=250), nullable=False)
10     size = Column(String(length=25))
11     price = Column(Integer)
12
13 engine = create_engine("sqlite:///sqlalchemy_example.db")
14
15 Base.metadata.create_all(engine)
```

Line 1 imports all the important creation items from the base SQLAlchemy module; these are used to create the database schema.

Line 2 imports the declarative base class, which our program will inherit from, after we create an instance of it in line 4.

The core of the declaration definitions is in lines 7-12. Each table in SQLAlchemy is defined as a class, and the parameters of the class are the individual columns that will make up the table. The columns are defined similar to the SQLite table, such as deciding the column that will be the primary key and the type of data contained within a column. When `nullable=False` is present, it means the column cannot be empty; otherwise, the column will assume no data is necessary.

Line 15 creates the DB engine that will handle the interaction with the actual DB file, and line 17 creates the tables from the defined classes. In this case, we only have one table, but large databases may have multiple ones that reference others.

When this program is run, a new database file (`sqlalchemy_example.db`) should appear within the current directory.

Filling and querying the database

Once the database is created, we can now populate it, as demonstrated in this code listing:

```
# sqlalchemy_db.py (part 1)
1 from sqlalchemy import create_engine
2 from sqlalchemy.orm import sessionmaker
3 from sqlalchemy_declarative import Tools, Base
4
5 engine = create_engine("sqlite:///sqlalchemy_example.db")
6 Base.metadata.bind = engine
7
8 DBSession = sessionmaker(bind=engine)
9 session = DBSession()
10
11 box_knife = Tools(name="Box Knife", size="Small", price=15)
12 drill = Tools(name="Drill", size="Medium", price=35)
13 axe = Tools(name="Axe", size="Large", price=55)
14 putty_knife = Tools(name="Putty Knife", size="Small", price=25)
15 hammer = Tools(name="Hammer", size="Small", price=25)
```

Continuing with reimplementing the SQLite example, we are filling the SQLAlchemy DB with the same tools inventory. Line 1 imports the DB engine module, while line 2 imports the session-making module. Line 4 imports the important classes from the previously created `sqlalchemy_declarative.py` file.

Lines 6 and 7 create the DB engine and bind it to the `Base` class so the declarative classes can be accessed by the session.

Lines 9 and 10 establish the communications with the database and provide the staging area for the objects that will populate the DB. Until the data is committed to the database, they are stored in the temporary session. If desired, the changes can be undone using `session.rollback()`.

Lines 11-15 provide the actual data that will be placed in the database. Each row of the database is represented by a separate variable definition:

```
# sqlalchemy_db.py (part 2)
1 screwdriver = Tools(name="Screwdriver", size="Small", price=10)
2 crowbar = Tools(name="Crowbar", size="Large", price=60)
3 items = (box_knife, drill, axe, putty_knife, hammer, screwdriver, crowbar)
4 session.add_all(items)
5 session.commit()
```

Here, line 3 creates a tuple of all the items to be added to the database, then line 4 uses the tuple to fill the DB with one command. Finally, we commit the changes to the database in line 5.

To retrieve the data from the database, we will use the following code:

```
# sqlalchemy_retrieval.py (part 1)
1 from sqlalchemy_declarative import Base, Tools
2 from sqlalchemy import create_engine
3 from sqlalchemy.orm import sessionmaker
4
5 engine = create_engine("sqlite:///sqlalchemy_example.db")
6
7 Base.metadata.bind = engine
```

```
8
9 DBSession = sessionmaker()
10 DBSession.bind = engine
11 session = DBSession()
```

We've seen lines 1-11 before, so we'll move to the main code in the following snippet:

```
# sqlalchemy_retrieval.py (part 2)
1 # Query all entries in database
2 tools = session.query(Tools).all()
3 for tool in tools:
4     print(tool.name)
5
6 # Return first entry in database
7 tool = session.query(Tools).first()
8 print("\n" + tool.name)
9
10 # Return the tool with given price
11 priced_tool = session.query(Tools).filter(Tools.price == 10).one()
12 print("\n" + priced_tool.name + "\n")
```

Line 2 queries the DB for all the entries within the database; since we only have one table, this is not a problem. Obviously, if you have a lot of data in a database, you wouldn't do this. With the query complete, we use a `for` loop to iterate over the returned list and print the names of the tools in lines 3 and 4.

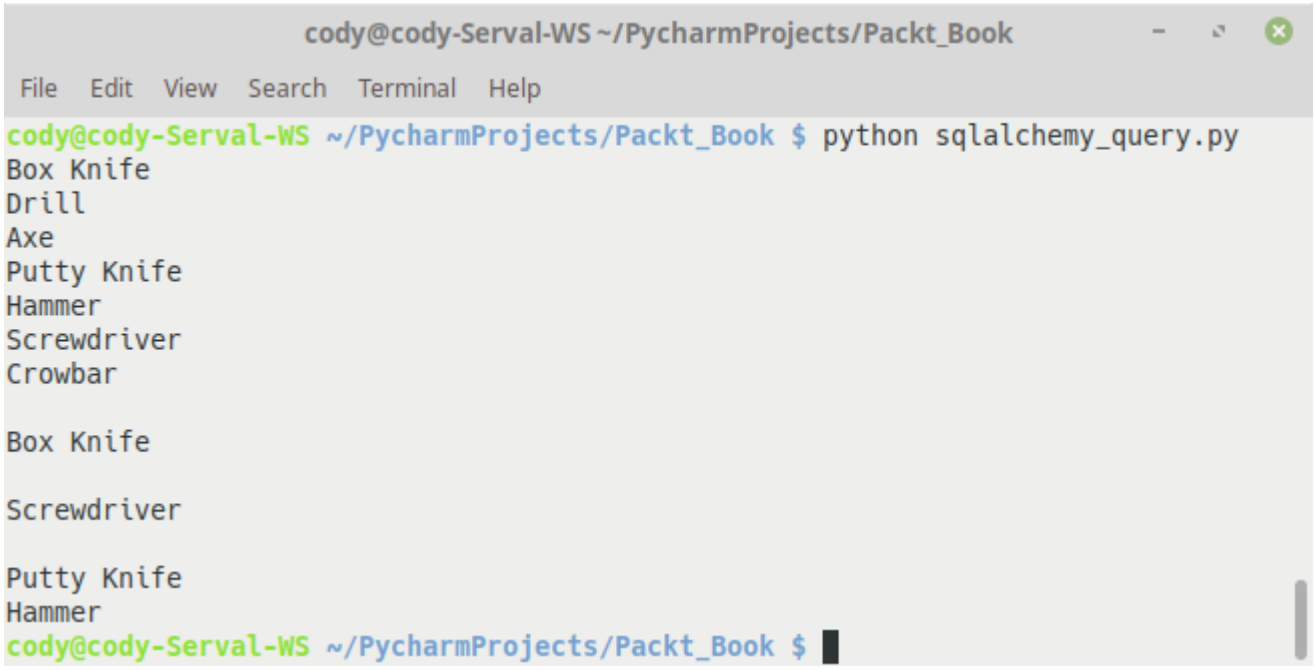
Line 7 does a similar query, but only returning the first entry in the database, which is subsequently printed in line 8. (To keep the final output separated for clarity, a newline character has been added.)

Line 11 performs a query that looks for the first item that matches the provided filter; in this case, we are looking for an item with a price = `10`. The item is then printed, separated from the other output:

```
# sqlalchemy_retrieval.py (part 3)
1 # Return all the tools with a given price
2 priced_tools = session.query(Tools).filter(Tools.price == 25).all()
3 for tool in priced_tools:
4     print(tool.name)
```

Line 2 looks for all entries that match a price of `25`. The resultant list is then iterated through to provide the name of the matching tools.

When this program is run, the results should look like the following screenshot:



SQLAlchemy query results

Obviously, there is more to DBs and SQLAlchemy than can be covered here. Hopefully, this provided you with an idea of how Python can be used to work with DBs, and some of the tools available depending on your needs.

From here, we will move into the main portion of this book: writing a program with real-world applications and significant complexity that, when finished, should give the reader confidence in being a software developer.

Summary

In this chapter, we discussed how Python works with OS filesystem I/O operations to interact with files, and how to work with DBs. Specifically, we looked at SQLite, which is included with Python, and how to use the Python-SQLite commands to use SQL queries. Finally, we reviewed some of the basics of SQLAlchemy, a DB framework that allows more Python-centric access to DB operations, without requiring the direct use of SQL statements.

In the next chapter, we will talk about application planning, including the software development life cycle, development best practices, determining project requirements, and establishing a software repository.

Application Planning

In the old days, software development was primarily dictated by management in terms of the programming languages and tools to be used, as well as providing the different resources necessary to complete the work.

Nowadays, companies and developers are more cognizant of privacy and security issues, as well as including user feedback early in the development process. Another key factor is the flexibility of development compared to system integration, developer efficiency, and user control of the software.

While this won't be a complete course in the administrative side of software development, this chapter will cover the following topics:

- Software development life cycle
- Development practices and methodologies
- Project requirements
- Software repositories

Software development life cycle

The development cycle divides software creation into several distinct phases, all focused on improving the design, product management, and project management processes of software engineering. Often, each phase culminates in a milestone, a designated deliverable, or some other marker of phase transition.

The software life cycle is a subset of systems development that was developed in the 1960s to create a formal framework for building information systems. As such, there have been numerous processes and methodologies created over the years to incorporate new technology, to address shortcomings in previous methods, because management demanded it, or other reasons.

Some of the different practices include the following:

- Structured programming
- Object-oriented programming
- Rapid application development
- Scrum
- Rational unified process
- Extreme programming
- Agile
- Pair programming
- Open source

As software development frameworks have evolved and changed in the last 50 years, there is no right method to use. Sometimes a hybrid construct will work best, though most software engineering nowadays uses some form of agile development.

Development practices and methodologies

As there are numerous ways to manage software development, we will only cover some of the most popular ones here.

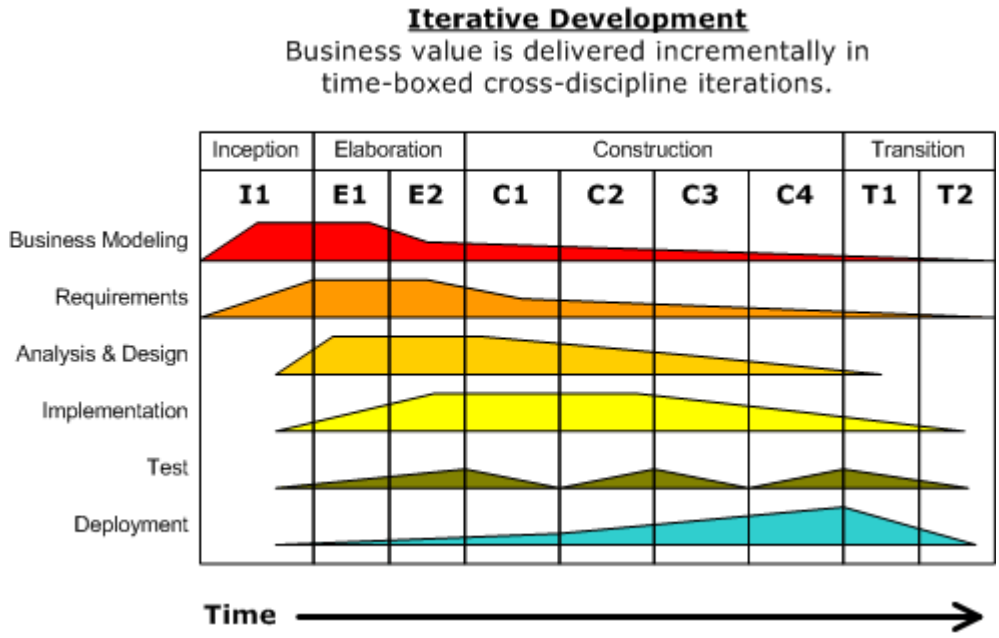
Incremental development

As the name suggests, **incremental** (or **iterative**) **development** (**ID**) is a succession of project segments that build on each other to create the final product. This allows for different aspects of the software to be fleshed out and tested prior to moving to the next step.

This doesn't mean that multiple steps can't be worked on simultaneously, just that a successive step isn't started until the previous one is judged to be complete. This development practice most aligns with traditional project management, where different aspects of the project can be developed at the same time, but within each aspect's pipeline, the steps are iterative and cumulative.

An example development process is shown in the following diagram. The columns represent the stages of the project, while the rows are the individual aspects. Each titled column has one or more sub-columns that represent the different milestones within each stage.

As can be seen in the following diagram, the areas of each colored graph move to the right as one moves down the chart. This makes sense, as initial development aligns with Requirements and Design, while finishing the project means deployment takes precedence. Testing cycles up and down during the project, as milestones are reached:



Incremental development process

Continuous integration

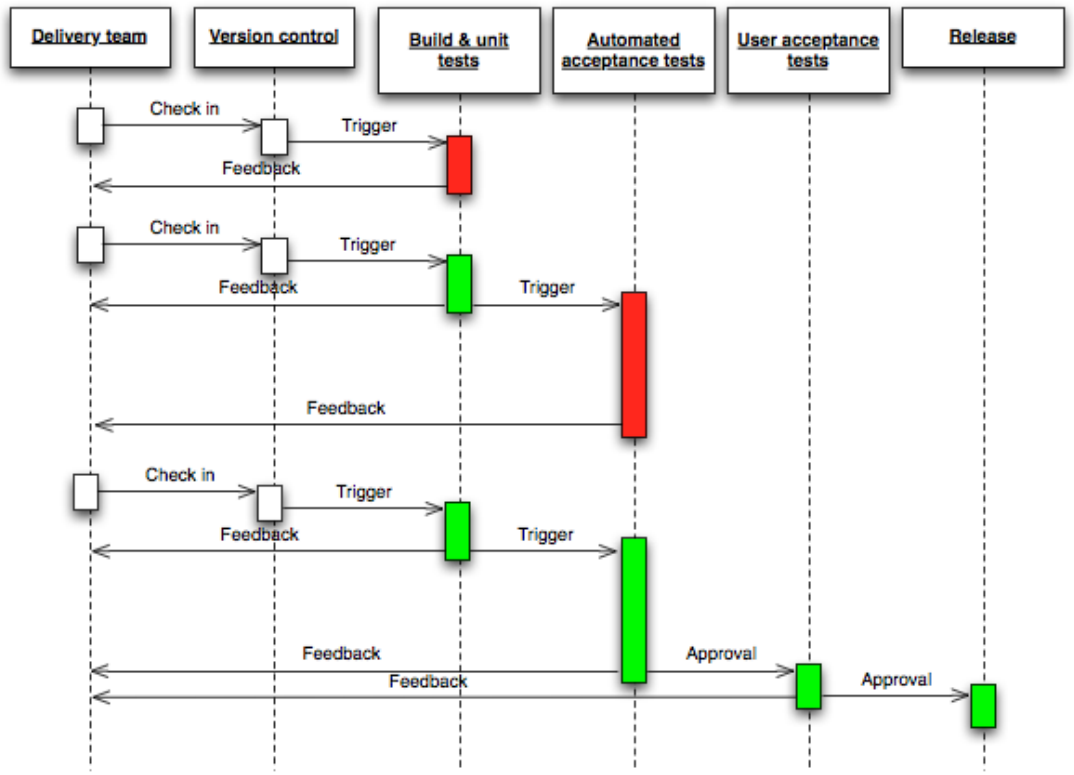
Continuous integration (**CI**) is the practice of merging all the code written by team members into a central repository, particularly into the main line of code. The continuous portion comes from the fact that these code merges occur multiple times per day, or at least once a day.

The main purpose of CI is to identify and remedy integration issues early in the development process. Programmers write unit tests for their code to ensure it works as expected. They can also check their work against the tests of other developers. When all the code is merged together, a build server will run automated tests to ensure compatibility between all submitted code. If any errors occur, they are identified immediately, rather than later on when it can be difficult to identify and fix them.

Another aspect of the build servers is to perform continuous quality control, such as profiling the software for performance bottlenecks, automatically creating documentation, and applying a variety of additional tests to the code base.

Development/**Operations** (**DevOps**) utilizes CI as the backbone of software creation. DevOps relies heavily on automated testing and monitoring of all steps of software development, with an eye toward reducing the development cycle, improving time to deploy, and making the final product more dependable and robust.

The CI process is demonstrated in the following diagram:



Continuous integration process

Attribution: CC BY-SA 1.0 (*Jez Humble* – http://continuousdelivery.com/2010/02/continuous-delivery/)

Prototyping

Software prototypes are simply models of the final product. They can be of nearly any part of the final application: user interface mock-ups, stand alone features, proof-of-concept, and so on.

Prototypes are intended to help guide the course of development, showing management and customers what the developers are working on, while allowing time for feedback before the programmers move forward. They are also useful when a large number of programmers are involved in a project, as each person or team can be responsible for a particular aspect of the final application. Their prototypes can be compared to other ones to ensure everyone knows what is going on with the project.

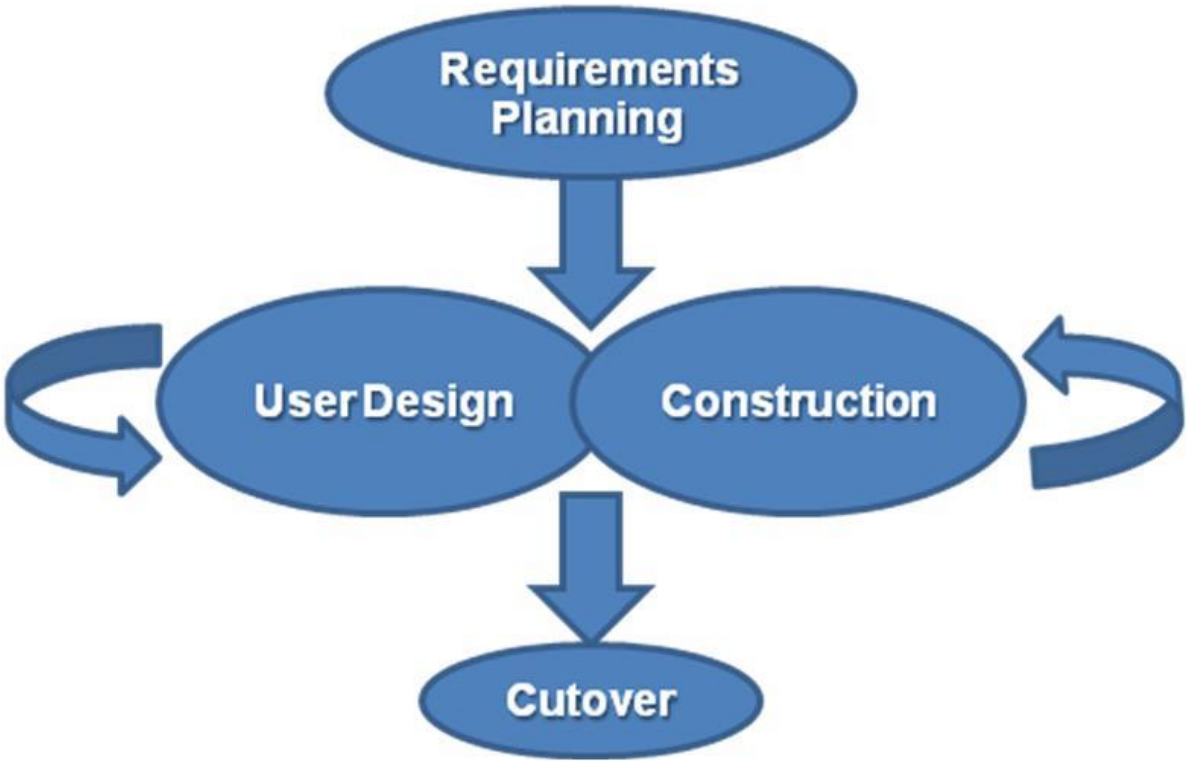
Rapid application development

Rapid application development (**RAD**) features aspects of iterative development and prototyping, focusing on rapid construction of prototypes to test out ideas and move the project forward, rather than spending a lot of time in the planning stage. In essence, the planning of the software is merged with the writing of the software, as planning and coding follow each other in a cyclical nature.

Of course, some roadmaps for the software exist. It's just that the pre-planning is minimized and, while the overall plan is followed, the path to get there may change during development. With the initial plan established, prototypes are created and used to dictate the plan for the next batch prototypes. This continues until the project is finished.

A chart of the RAD process is shown in the following diagram The initial planning stage moves into a development/design-plan cycle.

When the project is complete, it is cut over into the next phase, which may be testing, deployment, or another aspect of the overall project:



Rapid application development process

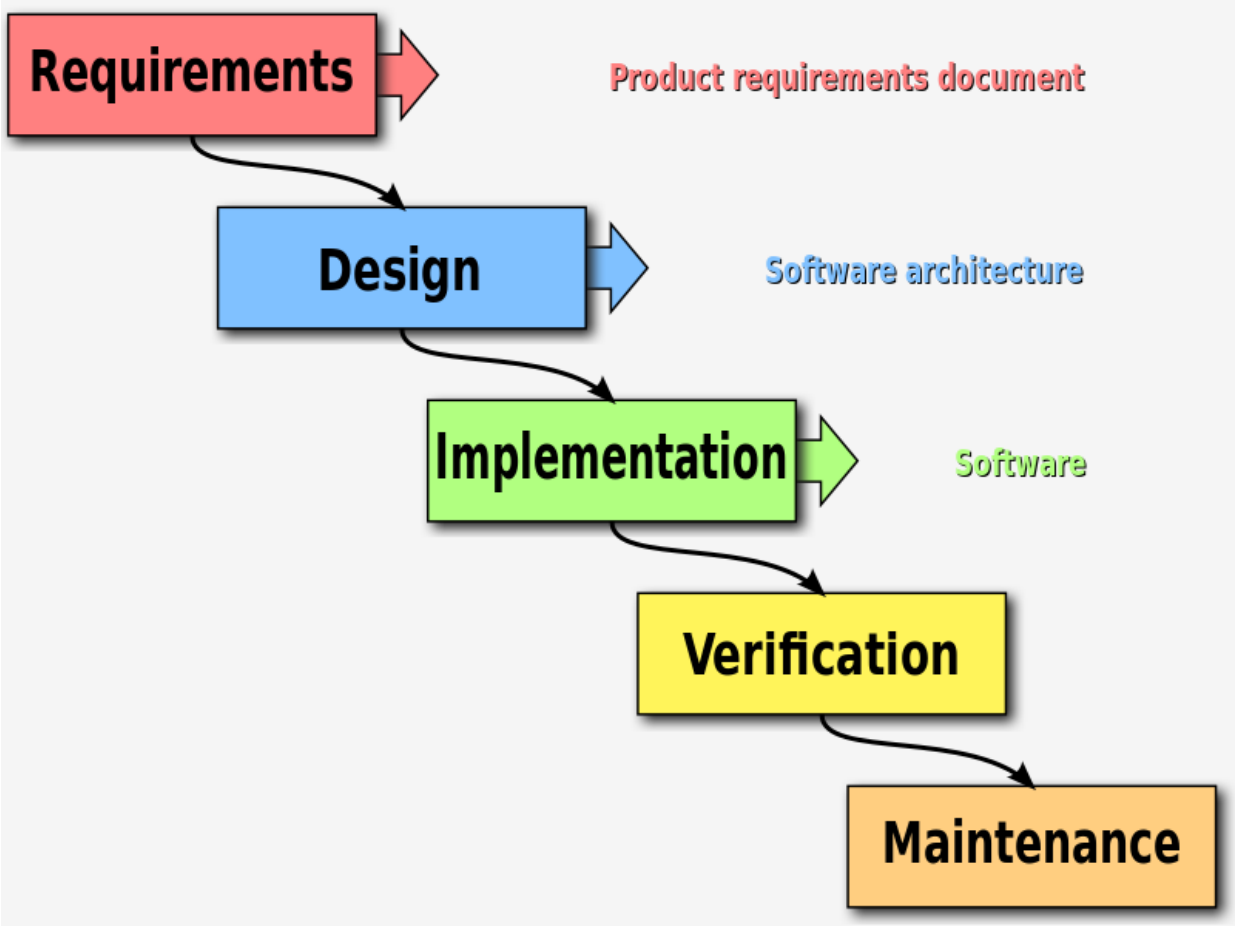
Waterfall development

Waterfall models were prominent in the early days of software development, being formalized into a standard during the mid-1980s by the US military. Waterfalls follow a sequential method, with development stages flowing downhill toward completion.

The stages are successive; though some allowance for overlap and rework is allowed, returning to a previous stage is discouraged. The main point to waterfall development is tight control of the entire project, from planning and maintaining deadlines to resource management and entire system implementation at one time. Project management focuses on the project lifetime through extensive documentation, formal reviews, and approval prior to commencing the next stage.

This approach is seen in classic project management models, regardless of the project, and is often seen as one reason why deadlines are missed, cost overruns occur, and the final product fails to deliver the expected results.

Another problem with waterfall development is that it is inflexible in terms of customer requirements. Once the project is approved, it continues until the product is delivered, regardless of whether the customer truly knows what that deliverable should actually do. The waterfall model is depicted in the following diagram:



Waterfall development
Attribution: CC BY 3.0 (*Peter Kemp* / *Paul Smith*)

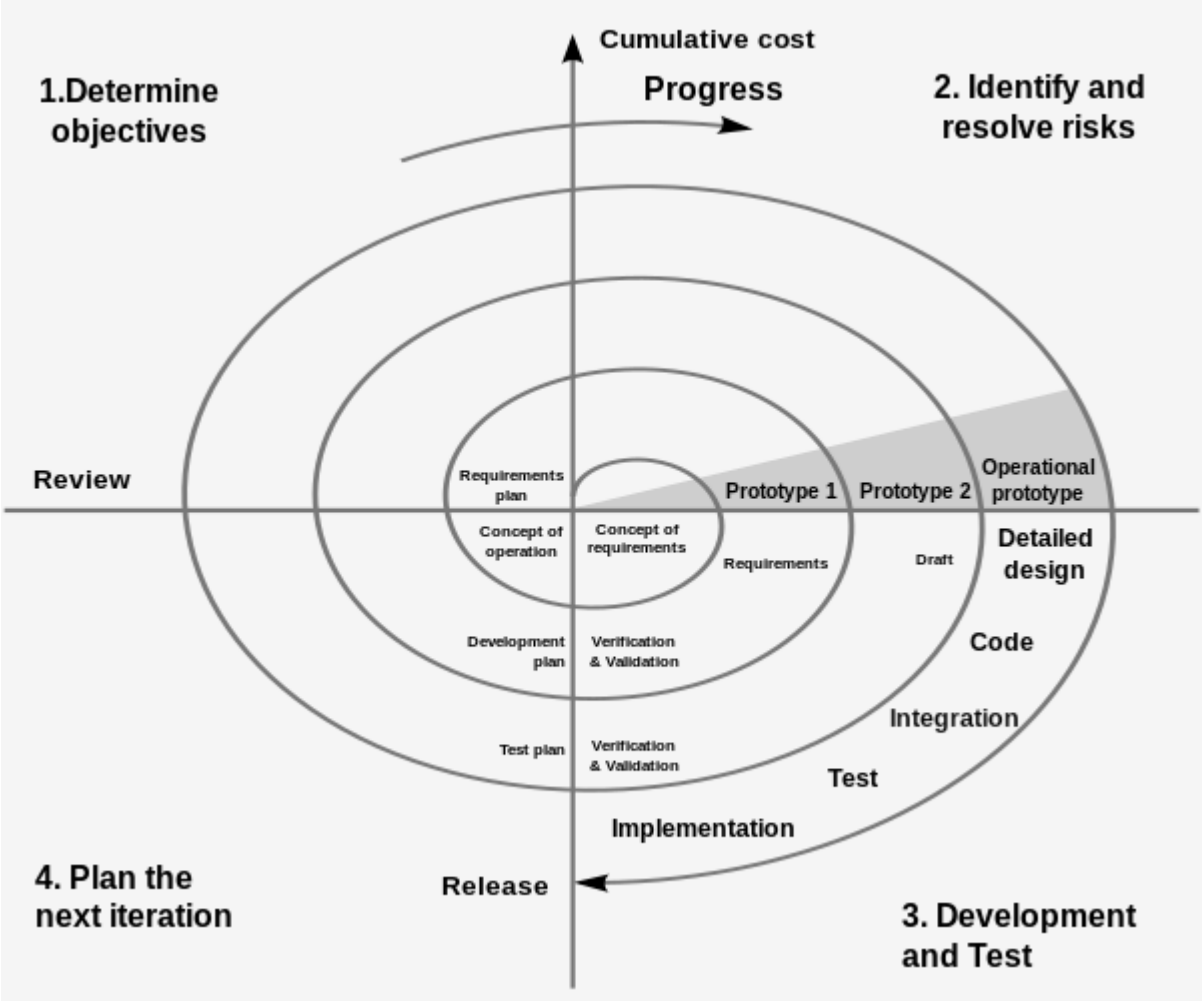
Spiral development

Spiral development aims to improve on the waterfall model by allowing for a repetitive development process. Like a spiral moving from the center outward, the process repeats each step as the project progresses, using the knowledge developed from the previous cycle to create each prototype.

Each loop covers the following items:

- Determine objectives and constraints for the new iteration
- Perform risk management and evaluate alternatives
- Develop deliverables for the current iteration
- Plan the next iteration

The spiral process is shown in the following diagram:



Spiral development

Agile development

Agile development is less of a process or methodology and more of a way of working for software developers. It focuses on adaptive planning, evolution of development, self-organized and cross-functional teams, continuous integration, and, above all, flexibility. The unofficial motto of agile development can summed up as "*release early*, *release often*" as stated by *Eric S. Raymond* in his seminal essay *The Cathedral, and The Bazaar*.

Though a variety of lightweight development methods and framework were in existence for many years, agile development could be said to have formally started in 2001, when the *Manifesto for Agile Software Development* was published. The Manifesto stipulated the following:

- People and their interactions are more important than tools and processes.
- Software should work as intended, while documentation should support the software. In other words, while comprehensive documentation is ideal, development should be applied toward the software, rather than documentation.
- Customer interaction is more important than contract negotiation.
- Adaptability and willingness to change are better than uncompromisingly sticking to a plan.

Based on these proclaimed values, the principles of agile development are as follows:

- Customer satisfaction through early and continuous delivery of software
- Welcoming changing requirements, even late in development
- Working software is delivered frequently; the shorter the time frame, the better
- Daily interaction between business people and developers
- Include motivated individuals in the project, and provide them with the tools and support necessary for success
- Face-to-face conversation is the best form of communication, both between developers and with leadership
- Working software is the primary measure of progress
- Sustainable development with the ability to maintain a constant pace
- Continuous attention to technical excellence and good design principles
- Simplicity, through the art of maximizing the amount of work not done, is essential
- The best architectures, requirements, and designs come from self-organizing teams
- On a regular basis, the development team reflects on how to become more effective, and adjusts their behavior accordingly

A number of agile development methods have developed over the years. While some concentrate on development practices, others focus on workflow management. Here is a list of some of the more popular practices and frameworks used in agile development:

- Scrum
- Kanban
- Extreme programming
- Feature-driven development
- Dynamic systems development method
- Agile testing
- Agile modeling
- Backlogs

- Behavior-driven development
- Continuous integration
- Pair programming
- Refactoring
- Retrospective
- Test-driven development
- User stories

Project requirements

Regardless of the design philosophy that will be used for the actual coding, the planning stage is perhaps the most important, at least when it comes to having a successful deployment. Many project management courses highlight planning as one of the key stages for any project; while agile development can mitigate poor planning, it doesn't completely eliminate all the problems.

Developing project requirements and evaluating them for incorporation is generally a job for the project or program manager, or whoever will be responsible for seeing the project to completion. A key point is that a project has a definite time frame associated with it; while work may be ongoing, a project has a designated start date and a completion deadline. This helps define what elements a project is supposed to have completed for the deliverable; anything beyond the scope of the project will be moved into a separate project.

Generally speaking, the key steps in developing a project are as follows:

1. Locate stakeholders, **subject-matter experts** (**SMEs**), and any relevant documentation and resources.
2. Identify the key problem(s) to be addressed.
3. Determine the scope of the project. This includes all the key features that will be included, as determined by the stakeholders. The scope needs to be reasonable for the given time frame, or the timeline extended.
4. Once the overarching scope has been approved, detailed requirements are developed and validated.
5. When the requirements are accepted, all stakeholders need to prioritize them, recognizing that low-priority ones may be pushed to the next version.
6. At this point, everything agreed upon should be formally documented to provide a roadmap for the project manager. This roadmap is vetted by stakeholders for final approval.
7. If the roadmap is approved, the project is frozen. Any additions or changes should be marked for inclusion in the next project, rather than modifying the current one. If the change is urgent and necessary, established change management processes should be used to document the change.
8. At this stage, the SMEs provide input regarding different options, solutions, risk factors, and costs. This input is incorporated into the development process.
9. The developers start work and provide regular updates to the project manager. This work continues until the project is finished, or at least this stage of a multi-phase project.
10. Any changes required are reviewed and assessed for relevance to the project's scope. If within the scope, the changes are documented via the change management process and incorporated into the project. If out of scope, the changes should be pushed to the next project.

There is much more to project management, as several certifications are available for the field, but this list should provide a general idea of the process.

The key takeaway from this section is that identifying and listing the requirements for a software project is important, as they guide the project from start to finish. Without clear requirements, the project can flounder.

An example from my personal experience should help clarify this. When placed in charge of a project at a new organization, it was discovered that the project had been in progress for 10 years and more than $1 million had already been spent. The project was to make a custom content management system (CMS) for the organization; the original designer wanted to use Python, so a third-party company was contracted to provide support to the in-house developers.

The problem was that the project was dictated by upper management and did not include the opinions or needs of the true users of the software. In addition, the requirements continually changed as leadership changed or new ideas were introduced. This was exacerbated by the fact that the users could rarely be bothered to test new versions of the software during development.

Thus, the developers were constantly trying to code for a moving target, and the contracted company didn't mind, as they were paid regardless. As a matter of fact, the company was able to use the knowledge from working on this project to create a brand-new version of their own CMS software.

Ultimately, I convinced management to abandon the project and find a commercial off-the-shelf product that met most of the needs of the users. That way, users could at least start working with something useful and request updates from the manufacturer as needed, rather than wait for vaporware that was never ready.

Software repositories

Software repositories, or repos, are storage locations for software, whether completed packages or code in development. Having a designated central location for software makes it easier for team members to access the code for development and referral. It can also be a location for end users to access the software.

Many software developers provide multiple versions of software, from nightly builds generated by a build server to various versions of the final product. Documentation is frequently included as well, often as separate files for downloading but sometimes as online HTML files.

For Python, the online PyPI repo is used for third-party Python modules; these are most commonly installed through the `pip` command. Another common option is to post packages onto GitHub; many packages available on PyPI have a hosting on GitHub as well.

GitHub is one of the most popular sites for open-source software, though Bitbucket, Gitlab, and other sites are available. That's not counting internal development repos for companies and organizations.

Software repositories often include, or provide extension support to, common build tools, such as CI build servers, documentation generators, automated testing suites, and so on. Often, these tools can be connected to a programmer's **integrated development environment** (**IDE**), such as PyCharm, allowing the programmer to rarely have to leave the IDE to use a tool.

Summary

In this chapter, we learned about some of the ways that project management in general, and software development specifically, can be used to help in the development of applications. We learned about the software development life cycle, a brief history of software project management and the various processes, methodologies, and frameworks available, the basic flow of assessing requirements and completing development, and how software repositories can be used to reduce the amount of administration in software projects.

In the next chapter, we will use this knowledge to make a plan of action for a liquid storage and transfer scenario. The requirements will dictate how detailed the program has to be, as well as how it will be utilized and what interface methods are required.

Writing the Imported Program

For this project, we will be creating a liquid storage and transfer system, suitable for nearly any liquid fluid scenario; while the term fluid can apply to both liquids and gases, we will only consider liquids in this book. In particular, we will include physics-based, mathematical formulas to calculate pressure and flow rate, while accounting for differences in liquid density to make the model suitable for any liquid.

Specifically, we will cover the following topics:

- Project requirements
- Utility functions
- Simulating storage tanks
- Simulating valves
- Simulating pumps

Project requirements

Because this was more of a personal project than a demand from a customer, the requirement assessment process was somewhat abbreviated. However, some of the needs were dictated by failures of previous tools, so a little background is in order.

A virtualized **Industrial Control System** (**ICS**) modeling program was provided by a vendor, but it never worked as desired. Two of the scenarios were irrelevant to the organization's mission, while the other two scenarios were broken: if a circuit breaker was opened, there were no cascading effects, like other circuit breakers opening. Therefore, when cyber security personnel were attempting to identify and protect critical components, there were no consequences felt within the system.

The physical ICS model used by the organization was ready for expansion, and one request was for a fuel storage and transfer system. With the knowledge what was lacking from the previous ICS application, I decided to make a fuel scenario that accounted for cascading effects, while providing alternative options for the security team.

A brief summary of the necessary requirements is listed here:

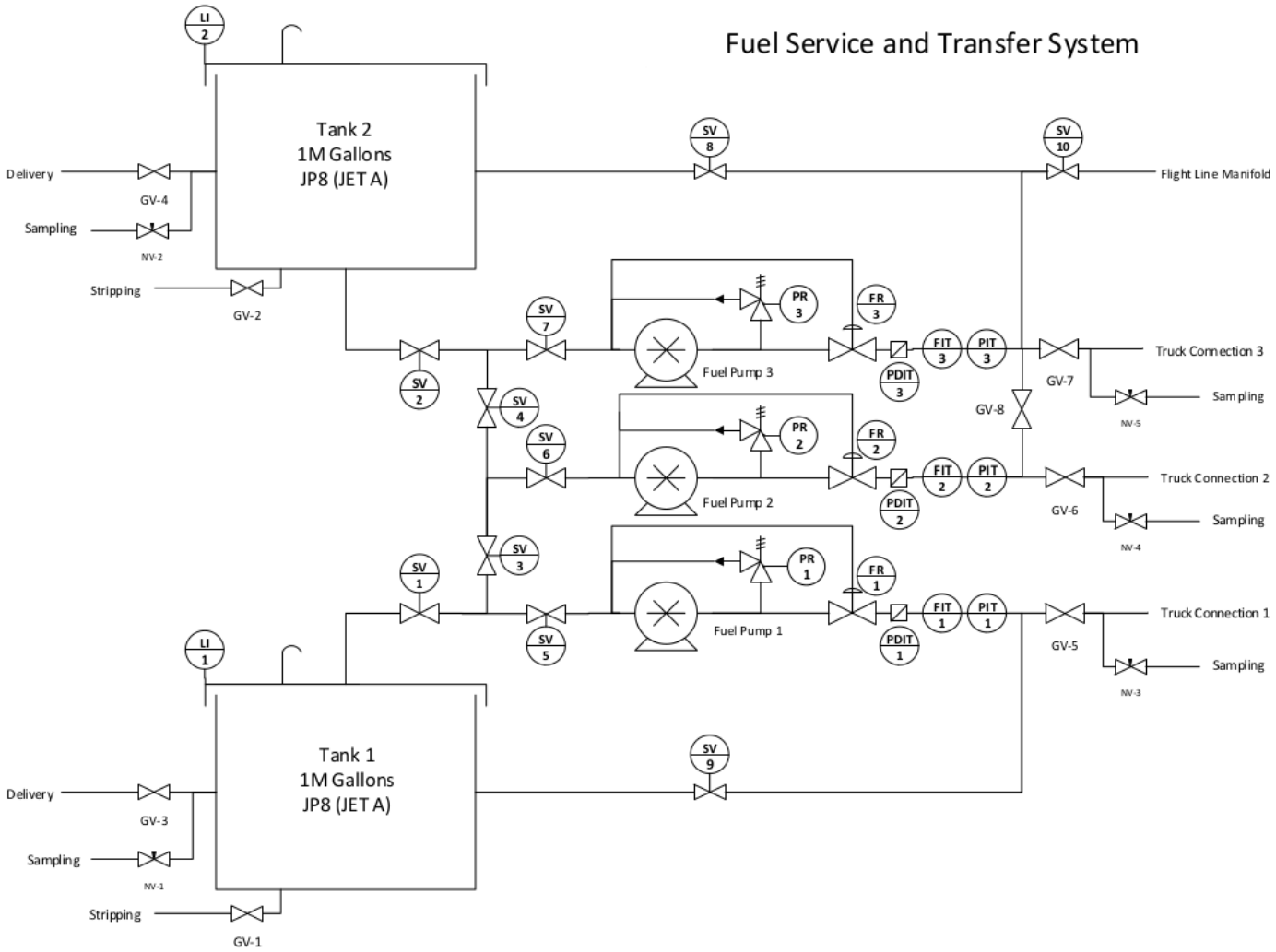
- General-use backend model was needed, one that can be used for any liquid, not just fuel. This means the same backend program can be used for fuel, water, chemicals, and so on.
- Physics-modeling of flow rates and pressure drops (within reason) was necessary. Because the cyber security teams wouldn't have the engineering background to notice invalid parameters, some of the components would be "idealized" to be frictionless or otherwise theoretical models. However, the main components, such as valves and pumps, needed to be modeled as realistically as possible, so cascading effects could be simulated.
- A graphical interface should be available. This could be the standard human-machine interface currently used with the physical model or a Python-centric GUI.

- Pure Python code would be used, but the ability to use ICS protocols, such as Modbus, should be supported.

One of the key points of this project, the physics-based modeling, could be considered unusual for software developers. Most software projects are business applications; programmers have little need to know about mathematical modeling or they have engineering teams who can provide the formulas. As I served in the Navy as a nuclear engineering laboratory technician, the necessary knowledge and skills were already available.

You are not expected to know the engineering principles that will be used for this scenario, but it will demonstrate some of the real-world applications of programming beyond making a website or other typical business applications.

The schematic diagram that will be used when designing this project is shown as follows:



Project schematic diagram

You don't have to understand the entire system, but a basic explanation follows, as the operations are part of the requirements development.

For this fuel storage and transfer systems, two 1-million gallon tanks store the fuel that is delivered by truck. The height of the fluid in the tanks provides a static hydraulic pressure to the supply lines going to the pumps.

The pumps are positive displacement, screw-type pumps that maintain a constant flow rate regardless of downstream pressure. Unlike centrifugal pumps, they do not suffer vapor lock if no fluid is present, as they can create their own suction. Also, unlike centrifugal pumps, they don't require a **Net Positive Suction Head** (**NPSH**) of incoming fluid. NPSH is the pressure of incoming fluid from static pressure due to gravity, as well as any additional pressure provided by other pumps. If the NPSH is too low for a given pump, the fluid will flash to vapor and cause damage to the pump, as well as stop the flow.

The valves are comprised of the following:

- There are gate valves, which are either fully open or fully closed. These are annotated as **SV** in the diagram.
- There are globe valves, which are designed to allow any position from fully open to fully closed. Because of their construction, they can throttle flow to any amount desired. These are annotated as **GV** in the diagram.
- There are pressure relief valves, which have a spring setting to ensure the downstream pressure from the pumps does not exceed parameters. If the pressure gets too high, the relief valve opens and sends the fuel back to the inlet of the pump. These are annotated as **PR** in the diagram.
- There are pressure regulating valves, which maintain a constant outlet pressure for the pumps. Regardless of the rest of the valves settings, the pump will continue to see the same pressure. These are annotated as **FR** in the diagram.
- There are needle valves, which are used for sampling fuel prior to use. These are annotated as **NV** in the diagram.

The circles with **FIT**, **PDIT**, and **PIT** are flow and pressure sensors, while **LI** is a tank level indicator. Not shown in the diagram are the check valves between the tanks and **SV-1** and**SV-2**, which prevent backflow into the tanks.

That's a lot of information, and most of it probably doesn't make much sense unless you have a mechanical background. Don't worry, as we will cover the most important information in the next few sections.

Utility functions

One thing that wasn't addressed in the requirements but will be necessary later is the need for utility functions. These are often found in applications and provide functionality for the main program, but don't directly apply to the core logic.

For this program, we will require a number of utility functions:

- Calculate liquid flow rate due to gravity
- Static hydraulic pressure of a fluid due to its height
- Conversion programs for pressure: specifically, we need to convert fluid head (measured in feet) in to **pounds per square inch** (**PSI**)

The functions are shown next (in separate parts), with explanations following each part:

```
# utility_formulas.py (part 1)
1 #!/usr/bin/env python3
2
3 import math
4
5 GRAVITY = 32.174  # ft/s^2
6 WATER_SPEC_WEIGHT = 62.4  # lb/ft^3
7 WATER_DENSITY = 1.94  # slugs/ft^3
8 WATER_SPEC_GRAV = 1.0
```

Line 1 is the "she-bang" line, telling *nix operating systems to use the default Python3 environment. Specifically, it tells the OS that the script can be executed without having to type `python` first at the command line.

Because we will be working with physics formulas, we will need to import the `math` module in line 3. However, we will be using basic algebraic formulas; we don't need to import any special physics modules.

Lines 5-8 define the constants that will be used in this file. Note that we are not using metric values; in the US, both metric and imperial values are used in engineering. However, it is often easier to locate imperial numbers than metric numbers for equipment. Hence, we arbitrarily selected imperial values and will convert from metric values when necessary:

```
# utility_formulas.py (part 2)
1 def gravity_flow_rate(diameter, slope, rough_coeff=140):
2     """Calculates approximate fluid flow due to gravity.
3
4     Should be within 5% of actual value.
5
6     Based on the Hazen-Williams equation (https://en.wikipedia.org/wiki/Hazen-
Williams_equation). Assumes a 2 inch, polyethylene pipe
7     """
8     coeff = math.pow(rough_coeff, 1.852)
9     diam = math.pow(diameter, 4.8704)
10    root_flow = math.sqrt(((coeff * diam * slope) / 4.52))
11    return root_flow
```

The first utility function is defined in `gravity_flow_rate()` on line 1. This function calculates a liquid's flow rate simply due to gravity. Because it doesn't have to be exact, the final result is within 5% of the real number. The formula requires a coefficient value based on the type of material the fluid is flowing through; the `rough_coeff` value is based on plastic piping but can be changed depending on the situation. The diameter of the pipe also plays a factor in flow rate; a larger diameter allows more fluid to flow without turbulence:

```
# utility_formulas.py (part 3)
1  def static_press(height, density=WATER_DENSITY):
2      """Calculate static pressure for any fluid"""
3      press = density * GRAVITY * height / 144
4      return press
5
6
7  def press_to_head(press, spec_grav=WATER_SPEC_GRAV):
8      """Calculate fluid head from pressure."""
9      head = (74.215 * press) / (spec_grav * GRAVITY)
10     return head
11
12
13  def head_to_press(head, spec_grav=WATER_SPEC_GRAV):
14      """Calculate pressure from fluid head."""
15      press = (spec_grav * GRAVITY * head) / 74.215
16      return press
```

The next function is defined as `static_press()` in line 1. Here, we calculate the static pressure of a column of fluid, that is, the pressure of a non-moving liquid. Because pressure in a liquid is felt equally throughout the fluid, and it cannot be compressed, it doesn't matter how large the container is, only the height of the fluid. Therefore, the static pressure of 15 feet of water behind a dam is the same as 15 feet of water in a vertical tube. For this calculation, we only need to know the density of the liquid and the height of the liquid, as we already have defined the gravitational constant.

Next, we convert from PSI in to feet of fluid head in line 7. This is a simple calculation that only requires the PSI value and the specific gravity of the fluid.

Conversely, in line 13, we convert from head in to PSI. Again, the only variables of concern are the specific gravity and fluid head (in feet):

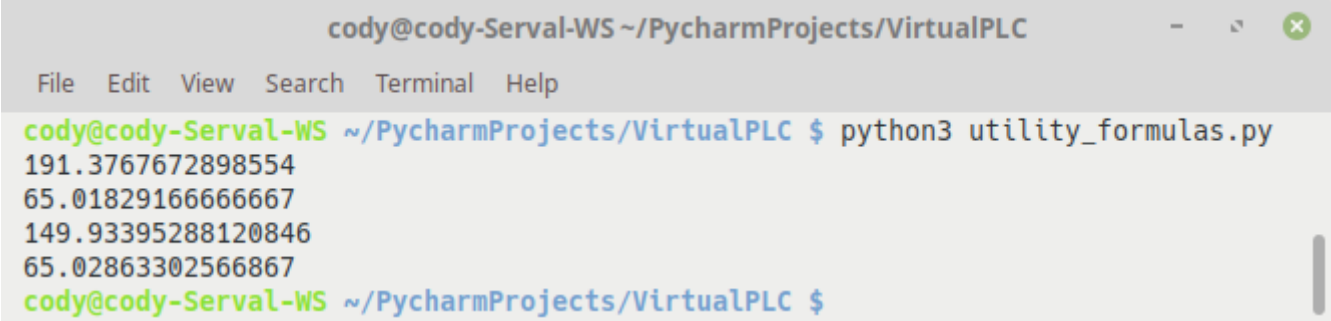
```
# utility_formulas.py (part 4)
1 if __name__ == "__main__":
2     print(gravity_flow_rate(2, 0.6))
3     print(static_press(150))
4     print(press_to_head(65.0))
5     print(head_to_press(150))
```

Finally, we have simple tests at the end of the file that provide a sanity check of these calculations. These results can be compared to hand-calculated results to see if the formulas work correctly.

This is also a good time to reiterate how line 1 works. The Python interpreter looks for the main program that will be running; each module within a program has its own identity name, with the main program's logic controlling the application's behavior. Therefore, if a particular module is given the name `__main__`, any code listed below this line will be run. If the module is imported into the main program, then any code below this line will not be run.

In other words, if `utility_formulas.py` is called directly by the Python interpreter (that is, it has the name `__main__`), then lines 2-5 will be processed. However, if `utility_formulas.py` is imported into another module, then lines 2-5 will be ignored.

The output of running the code on its own is shown in following screenshot:



Utility functions output

Simulating storage tanks

Now that we have our utility formulas defined, we can move into modeling the environment. First, we will figure out how to write the storage tanks, as they are fairly simple constructs: all we need to figure out is the pressure of the fluid in the tanks, as well as the flow rate out due to gravity. In addition, we need to know the tank level, as that affects the pressure of the fluid.

The code for the tanks is shown next, followed by explanations:

```
# tank.py (part 1)
1  #!/usr/bin/env python3
2  """
3  VirtualPLC tank.py
4
5  Purpose: Creates a generic Tank class for PLC-controlled SCADA systems.
6
7  Author: Cody Jackson
8
9  Date: 5/28/18
10 ################################
11 Version 0.2
12    Added path extension to alleviate errors
13 Version 0.1
14     Initial build
15 """
```

Line 1 is the "she-bang" command for the OS. Lines 2-15 provide a brief description of the program and its version history:

```
# tank.py (part 2)
1  import sys
2  sys.path.extend(["/home/cody/PycharmProjects/VirtualPLC"])
```

```
3  from Utilities import utility_formulas
4  import numbers
5
6  class Tank:
7      """Generic storage tank."""
8      def __init__(self, name="", level=0.0, fluid_density=1.94, spec_gravity=1.0,
outlet_diam=0.0, outlet_slope=0.0):
9          self.name = name
10         self.__level = float(level)  # feet
11         self.fluid_density = fluid_density  # slugs/ft3
12         self.spec_grav = spec_gravity
13         self.__tank_press = 0.0
14         self.flow_out = 0.0
15         self.pipe_diam = outlet_diam
16         self.pipe_slope = outlet_slope
17         self.pipe_coeff = 140
```

In this part, with lines 1 and 2, we import the `sys` module and use it to extend the system path (the locations where files are looked for) to add the project's root directory. Without this, the importation of some of the project modules, such as the utility formulas, will error out, as Python doesn't know exactly where they are located.

In lines 3 and 4, we import the previously written utility formulas, as well as the built-in `numbers` module. The `numbers` module provides for a hierarchy of abstract base classes of numeric objects. While it provides a number of useful features, the main thing we'll use it for in this program is to simply determine whether an argument is a number or not.

In line 6, we define our `Tank` class. Lines 9-17 define the parameters that will be used when an instance of `Tank` is created. The parameters defined are the following:

- The tank instance name
- The fluid level in the tank
- The fluid's density and specific gravity, with default values for water
- The fluid pressure as measured at the bottom of the tank (the static pressure)
- The gravity flow rate of the fluid
- The outlet pipe's diameter, slope, and roughness coefficient

The next code listings are all part of the `Tank` class, until we reach `if __name__ == "__main__"`:

```
# tank.py (part 3)
1    @property
2    def static_tank_press(self):
3        """Return hydrostatic tank pressure."""
4        return self.__tank_press
5
6    @static_tank_press.setter
7    def static_tank_press(self, level):
8        """Calculate the static fluid pressure based on tank level."""
9        try:
10           if not isinstance(level, numbers.Number):
11               raise TypeError("Numeric values only.")
12           elif level <= 0:
13               self.__tank_press = 0.0
14           else:
15               self.__tank_press = utility_formulas.static_press(self.level,
self.fluid_density)
16       except TypeError:
17           raise  # Re-raise for testing
```

Starting with line 1, we define the static pressure property. With a Python property, whenever the specific variable is called without arguments, the current value of the variable is returned. To set this variable, we use the setter method in lines 6-17.

The setter method takes the tank level as an input argument and checks to ensure it is actually a number. If not, then an error is generated. Next, if the level is zero or less, the tank pressure is set to zero. Finally, if there is a fluid level in the tank, the pressure for that fluid level is calculated. Note that no value is returned; a setter method simply sets the value for a property but doesn't return anything:

```
# tank.py (part 4)
1    @property
2    def level(self):
3        """Return fluid level in tank."""
4        return self.__level
5
6    @level.setter
7    def level(self, level):
8        """Set the level in the tank."""
9        try:
10            if not isinstance(level, numbers.Number):
11                raise TypeError("Numeric values only.")
12            elif level <= 0:
13                self.__level = 0.0
14            else:
15                self.__level = level
16        except TypeError:
17            raise  # Re-raise error for testing
18        finally:
19            self.static_tank_press = self.level
20            self.gravity_flow(self.pipe_diam, self.pipe_slope, self.pipe_coeff)
```

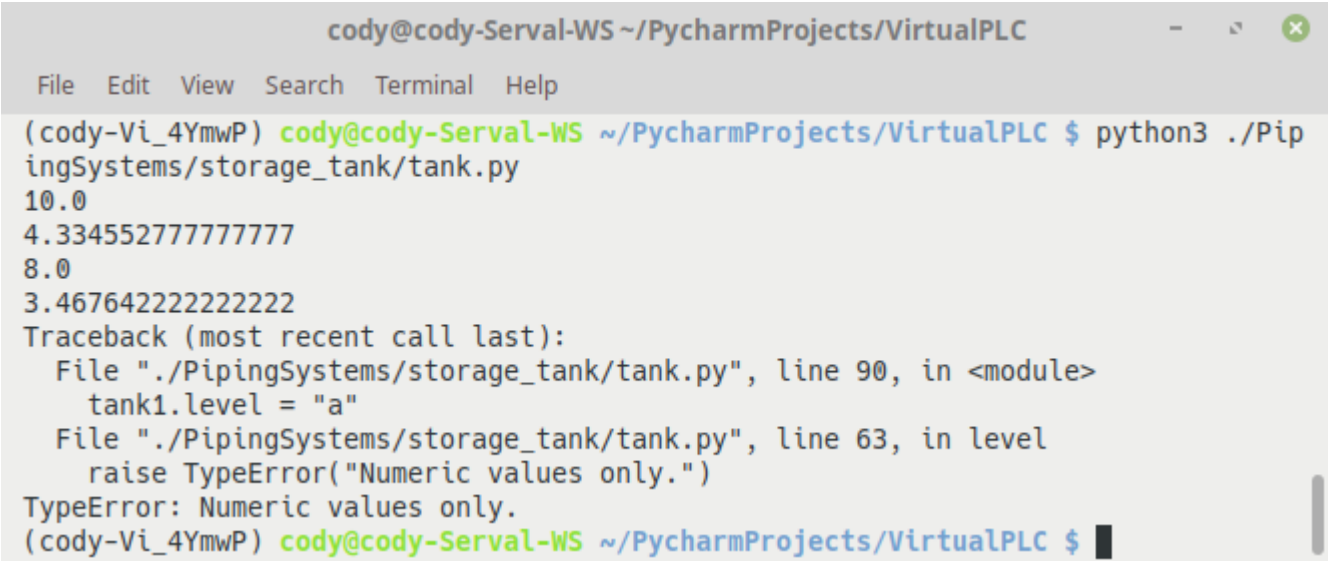
The preceding code listing creates the property and setter method for tank level. The operations are similar to fluid pressure, except a `finally` statement is provided to pass the tank level into the pressure setter, as well as calculate the gravitational flow rate based on the level:

```
# tank.py (part 5)
1    def gravity_flow(self, diameter, slope, pipe_coeff):
2        if self.level > 0:
3            self.flow_out = utility_formulas.gravity_flow_rate(diameter, slope,
pipe_coeff)
4        else:
5            self.flow_out = 0.0
6
7
8 if __name__ == "__main__":
9     tank1 = Tank("tank1", 10)
10    print(tank1.level)
11    tank1.static_tank_press = tank1.level
12    print(tank1.static_tank_press)
13    tank1.level = 8.0
14    print(tank1.level)
15    tank1.static_tank_press = tank1.level
16    print(tank1.static_tank_press)
17    tank1.level = "a"
18    print(tank1.level)
```

In the preceding code listing, lines 1-5 comprise the method to calculate the gravitational flow rate. First, it checks that the tank level is greater than zero and, if true, it calculates the flow rate. Otherwise, it sets the flow rate to zero.

Finally, we have the simple tests in lines 8-18. These are not full-blown unit tests, but verification tests while writing the code.

The output of directly running the tank code is shown in following screenshot:



Tank code output

Name mangling

You'll notice that some of the variable names have leading double underscores. This is to define them as specifically related to this class. This is referred to as **name mangling** in Python.

In Python, a single, leading underscore creates a so-called `private` attribute or method, designed to keep the attribute from being directly accessed. It's not really private, as it can be called explicitly, but it won't be automatically imported when using the `import` statement.

Name mangling causes the attribute with double underscores to become `_ClassName__attribute`internally within Python. That is, the class name the attribute is associated with becomes part of the fully qualified name for the attribute.

Name mangling is intended to provide private instance variables and methods to classes without the programmer having to worry about name shadowing or other attributes with the exact name.

While not necessary, using name mangling for property variables is an easy way to ensure properties are returned or set for the desired object, without having to worry about capturing the wrong one.

Simulating valves

Valves and pumps are the key components of most piping systems, as they affect how fluid flows through the system and does work. We will first talk about modeling valves; since there is a lot of information in this code, we will break it into separate subsections.

Base valve class

Before we can do anything with individual valves, we have to create a basic `Valve` class that the unique valves will inherit from. This base class will have to have as many characteristics as each valve type will need to make the base class as generic as possible. The base valve class will be broken up into multiple code listings:

```
# valve.py (part 1)
1  #!/usr/bin/env python3
2  """
3  VirtualPLC valve.py
4
5  Purpose: Creates a generic Valve class for PLC-controlled SCADA systems.
6
7  Classes:
8      Valve: Generic superclass
9      Gate: Valve subclass; provides for an open/close valve
10      Globe: Valve subclass; provides for a throttling valve
11      Relief: Valve subclass; provides for a pressure-operated open/close valve
12
13  Author: Cody Jackson
14
15  Date: 4/9/18
16  ################################
17  Version 0.1
18      Initial build
19  """
```

We've seen lines 1-19 before in the `tank.py` program; they simply provide some basic information about the program.

```
# valve.py (part 2)
1  import math
2
3  class Valve:
4      """Generic class for valves.
5
6      Cv is the valve flow coefficient: number of gallons per minute at 60F through a
fully open valve with a press. drop of 1 psi. For valves 1 inch or less in diameter,
Cv is typically < 5.
7      """
8          def __init__(self, name="", sys_flow_in=0.0, sys_flow_out=0.0, drop=0.0,
position=0, flow_coeff=0.0, press_in=0.0):
9           """Initialize valve."""
10          self.name = name
11          self.__position = int(position)  # Truncate float values for ease of
calculations
```

```
12          self.Cv = float(flow_coeff)
13          self.flow_in = float(sys_flow_in)
14          self.deltaP = float(drop)
15          self.flow_out = float(sys_flow_out)
16          self.press_out = 0.0
17          self.press_in = press_in
```

For this file, we're also importing the `math` module in line 1. Line 3 is where we start the generic base `Valve` class. The docstring in lines 4-7 provides some basic information to the user of the program. Lines 8-17 define the initialization parameters for a valve. Specifically, we set the following:

- We set the valve's name; not required but helpful when attempting to identify valves programatically.
- We set the valve's position as an integer, where `0` = closed and `100` = fully open. An integer value allows us to show the throttling value for a valve, such as 30% open.
- We set the valve's flow coefficient (`Cv`), which affects the pressure drop seen across the valve from inlet to outlet.
- We set the fluid flow rate into and out of the valve.
- We set the pressure drop across the valve.
- We set the fluid pressure coming into and leaving the valve.

Following code snippet demonstrates `valve.py` (part 3):

```
# valve.py (part 3)
1     def calc_coeff(self, diameter):
2         """Roughly calculate Cv based on valve diameter."""
3         self.Cv = 15 * math.pow(diameter, 2)
4
5     def press_drop(self, flow_out, spec_grav=1.0):
6         """Calculate the pressure drop across a valve, given a flow rate.
7
8         Pressure drop = ((system flow rate / valve coefficient) ** 2) * spec. gravity
of fluid Cv of valve and flow rate of system must be known.
9
10         Specific gravity of water is 1.
11               """
12         try:
13             x = (flow_out / self.Cv)
14             self.deltaP = math.pow(x, 2) * spec_grav
15         except ZeroDivisionError:
16             return "The valve coefficient must be > 0."
```

Line 1 defines a method to calculate the valve's flow coefficient. Normally, this value is provided by the manufacturer, but for simulations that don't require exact values, this method allows us to approximate Cv based on the diameter of the valve.

Line 5 is the method to calculate the pressure drop across a valve. The pressure across the valve is a function of the flow rate and Cv. It's not as simple as using Bernoulli's equation, as the flow coefficient is dependent on the valve structure; a gate valve has a much higher Cv compared to a globe valve:

```
# valve.py (part 4)
1     def valve_flow_out(self, flow_coeff, press_drop, spec_grav=1.0):
2         """Calculate the system flow rate through a valve, given a pressure drop.
3
4         Flow rate = valve coefficient / sqrt(spec. grav. / press. drop)
5               """
6         try:
7             if flow_coeff <= 0 or press_drop <= 0:
```

```
8                raise ValueError("Input values must be > 0.")
9            else:
10                x = spec_grav / press_drop
11                self.flow_out = flow_coeff / math.sqrt(x)
12                return self.flow_out
13        except ValueError:
14            raise  # Re-raise error for testing
```

Line 1 defines a method to calculate the outlet flow from a valve. This flow is dependent on the valve's Cv, the pressure drop across the valve, and the specific gravity of the fluid. The method ensures that both Cv and pressure drop are not less than 1; otherwise, an error is generated.

An interesting exception use is shown in lines 13 and 14. `ValueError` was raised in line 8 and has an error message that will be printed to the console when the exception is thrown. However, while it is caught in line 13, it is re-raised in line 14. Re-raising the error isn't normally necessary, but when we write unit tests, it is a useful characteristic to test for, as it ensures we are capturing the correct exception. As we haven't talked about testing yet, feel free to ignore this for now:

```
# valve.py (part 5)
1    def get_press_out(self, press_in):
2        """Get the valve outlet pressure, calculated from inlet pressure."""
3        if press_in:
4            self.press_in = press_in  # In case the valve initialization didn't
include it, or the value has changed
5        self.press_drop(self.flow_out)
6        self.press_out = self.press_in - self.deltaP
7
8    @property
9    def position(self):
10        """Get position of valve, in percent open."""
11        return self.__position
```

Line 1 is the start of the method to calculate outlet pressure. The method is "intelligent" enough to check for the inlet pressure and reset it to the current value; this ensures the calculation is correct, based on the current system operating parameters.

Line 8 is the start of the property method that provides for the valve's position, a measure of percentage open:

```
# valve.py (part 6)
1    @position.setter
2    def position(self, new_position):
3        """Change the valve's position.
4
5        If new position is not an integer, an error is raised.
6               """
7        try:
8            if type(new_position) != int:
9                raise TypeError("Integer values only.")
10            else:
11                self.__position = new_position
12        except TypeError:
13            raise  # Re-raise for testing
```

The preceding code listing is the start of the valve position setter method. It checks to ensure that the value input is an integer and notifies the user if not. We could have just truncated floating-point numbers, but this provides a little bit of backup to remind the user to double-check the values being input:

```
# valve.py (part 7)
1    def open(self):
2        """Open the valve"""
3        self.__position = 100
4        self.flow_out = self.flow_in
5        self.press_out = self.press_in
6
7    def close(self):
8        """Close the valve"""
9        self.__position = 0
10       self.flow_out = 0
11       self.press_out = 0
12       self.deltaP = 0
```

Lines 1-12 provide easy ways to set a valve to be fully open or closed, as they not only set the position value but also set the outlet flow rates and pressure values for the valve at the same time.

This is it for the parent valve class. The following subsections will describe the code for specific valve children.

Gate valve class

Gate valves are children of the parent `Valve` class, as shown in the following code example:

```
# valve.py (part 8)
1  class Gate(Valve):
2      """Open/closed valve."""
3      def read_position(self):
4          """Identify the position of the valve."""
5          if self.position == 0:
6              return "{name} is closed.".format(name=self.name)
7          elif self.position == 100:
8              return "{name} is open.".format(name=self.name)
9          else:  # bad condition
10              return "Warning! {name} is partially open.".format(name=self.name)
11
12      def turn_handle(self, new_position):
13          """Change the status of the valve."""
14          if new_position == 0:
15              self.close()
16          elif new_position == 100:
17              self.open()
18          else:  # Shouldn't get here
19              return "Warning: Invalid valve position."
```

The `Gate` subclass inherits everything from the `Valve` parent class. The only new methods it provides are `read_position()` and `turn_handle()`. As a gate valve is either open or closed (it doesn't affect fluid flow until more than 70% closed), we only need to know whether the valve is fully open or closed. In addition, when operating the valve, we only need to open or close it, so `turn_handle()` only accepts those two values; all others result in an error.

Globe valve class

Globe valves are also children of the `Valve` parent, as shown in the following code:

```
# valve.py (part 9)
1  class Globe(Valve):
2      """Throttling valve."""
3
4      def read_position(self):
5          """Identify the position of the valve."""
6          return "{name} is {position}% open.".format(name=self.name,
position=self.position)
7
8      def turn_handle(self, new_position):
9          """Change the status of the valve."""
10          if new_position == 100:
11              self.open()
12          elif new_position == 0:
13              self.close()
14          else:
15              self.position = new_position
16              self.flow_out = self.flow_in * self.position / 100
17              self.press_drop(self.flow_out)
18              self.get_press_out(self.press_in)
```

Globe valves can be any position from fully open to fully closed, so the `Globe` subclass is similar to the `Gate` subclass. The primary difference is in `turn_handle()`, which accepts any value from `0` to `100`. If the valve isn't fully open or closed, then the pressure and flow parameters of the valve are automatically adjusted.

Relief valve class

Relief valves are the last subclass of the `Valve` parent class, as demonstrated in the following code:

```
# valve.py (part 10)
1  class Relief(Valve):
2      """Pressure relieving valve.
3
4      Assumes full open when open set point reached and fully closed when close set
point reached. Does not affect flow rate or system pressure; those parameters must be
adjusted elsewhere.
5       """
6      def __init__(self, name="", sys_flow_in=0.0, sys_flow_out=0.0, drop=0.0,
position=0, flow_coeff=0.0, press_in=0.0, open_press=0, close_press=0):
7          """Inherits base initialization and adds valve open/close pressure
values."""
8          super(Relief, self).__init__(name, sys_flow_in, sys_flow_out, drop,
position, flow_coeff, press_in)
9          self.setpoint_open = open_press
10         self.setpoint_close = close_press
```

Pressure valves have pressure setpoints, above which the valve will open and below which the valve will close. There's actually a small pressure range between initial lifting and being fully open, as well as when closing. For our purposes, we don't need to account for that and simply model the valve to be fully open/closed when the setpoints are reached.

The `Relief` subclass defines these setpoints during initialization. This method is unique, compared to the other classes: it includes a `super()` call. This call ensures that the parent class, `Valve`, is called for all standard parameters, while allowing `Relief` to add the new parameters `setpoint_open` and `setpoint_close`. Hence, we don't have to redefine all of the initial parameters just to add the two new ones.

The methods used for the relief valve, shown in the following two code blocks, are similar to what we've seen before, so we won't elaborate on them:

```
# valve.py (part 11)
1    def read_position(self):
2        """Identify the status of the valve."""
3        if self.position == 0:
4            return "{name} is closed.".format(name=self.name)
5        elif self.position == 100:
6            return "{name} is open.".format(name=self.name)
7        else:   # bad condition
8            return "Warning! {name} is partially open.".format(name=self.name)
9â€‹
10    def set_open_pressure(self, open_set):
11        """Set the pressure setpoint where the valve opens."""
12        self.setpoint_open = open_set
13â€‹
14    def read_open_pressure(self):
15        """Read the high pressure setpoint."""
16        return self.setpoint_open
```

The following code snippet is part 12 of `valve.py`:

```
# valve.py (part 12)
1    def read_close_pressure(self):
2        """Read the low pressure setpoint."""
3        return self.setpoint_close
4
5    def set_close_press(self, close_set):
6        """Set the pressure setpoint where the valve closes."""
7        self.setpoint_close = close_set
8
9    def valve_operation(self, press_in):
10        """Open the valve if pressure is too high; close the valve when pressure
lowers."""
11        if press_in >= self.setpoint_open:
12            self.open()
13        elif press_in <= self.setpoint_close:
14            self.close()
```

The following code shows the simple tests for the valve program, much like we've done before with the utility functions and tank program. They are broken up into three parts, with each one dedicated to testing a specific child class type:

```
# valve.py (part 13)
1 if __name__ == "__main__":
2     # Functional test_valves
3     # name="", sys_flow_in=0.0, position=0, flow_coeff=0.0, drop=0.0, open_press=0,
close_press=0
4     gate1 = Gate("Pump inlet")
5     print("{} created".format(gate1.name))
6     print(gate1.read_position())
7     gate1.turn_handle(100)
8     print(gate1.read_position())
9     gate1.close()
10    print(gate1.read_position())
11    gate1.open()
12    print(gate1.read_position())
```

The following code snippet is part 14 of `valve.py`:

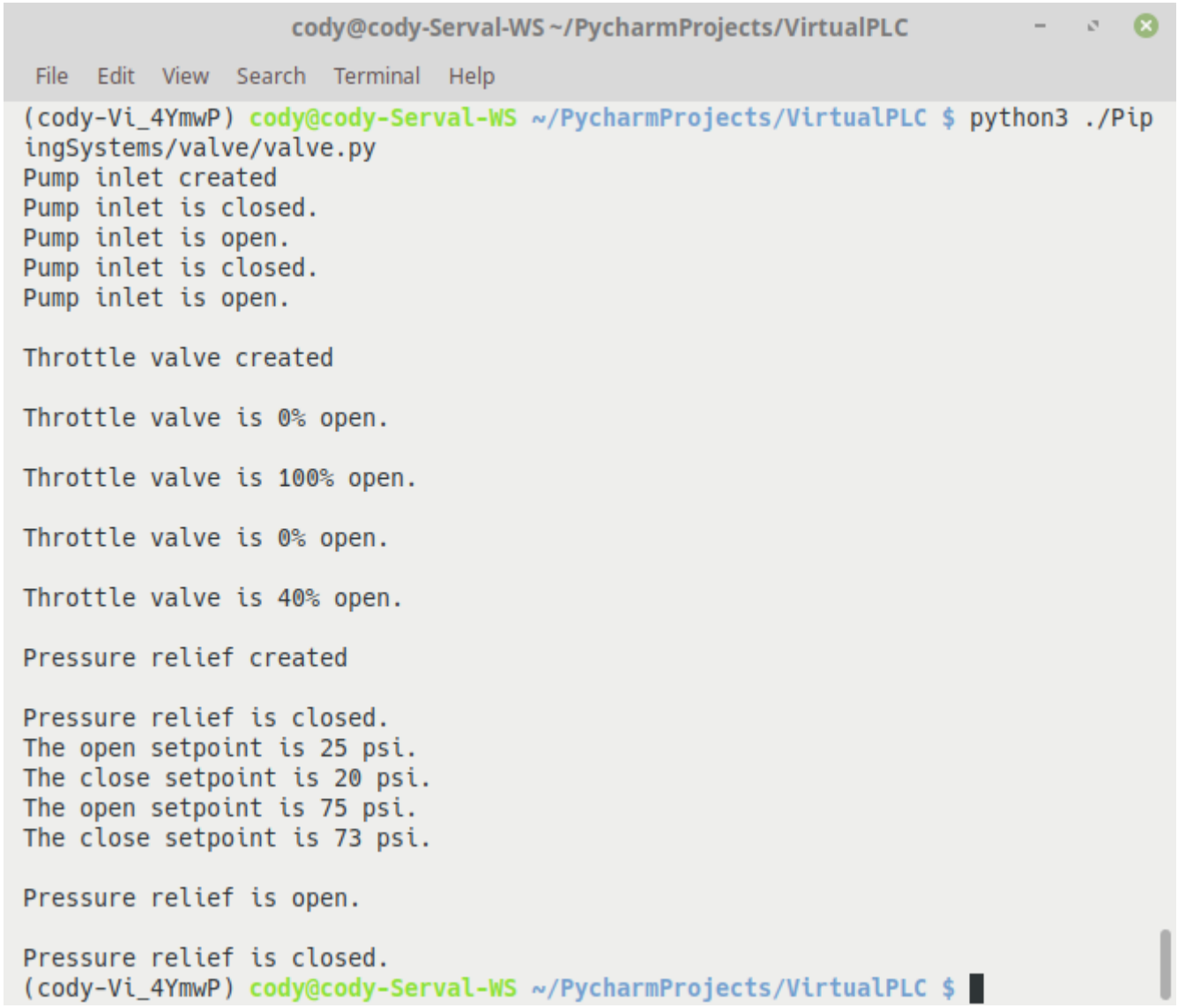
```
#  valve.py (part 14)
   globe1 = Globe("\nThrottle valve", flow_coeff=21, press_in=10, sys_flow_in=15)
    print("{} created".format(globe1.name))
    print(globe1.read_position())
    globe1.open()
    print(globe1.read_position())
    globe1.close()
    print(globe1.read_position())
    globe1.turn_handle(40)
    print(globe1.read_position())
```

The following code snippet is part 15 of `valve.py`:

```
#   valve.py (part 15)
 relief1 = Relief("\nPressure relief", open_press=25, close_press=20)
    print("{} created".format(relief1.name))
    print(relief1.read_position())
    print("The open setpoint is {} psi.".format(relief1.read_open_pressure()))
    print("The close setpoint is {} psi.".format(relief1.read_close_pressure()))
    relief1.set_open_pressure(75)
    relief1.set_close_press(73)
    print("The open setpoint is {} psi.".format(relief1.read_open_pressure()))
    print("The close setpoint is {} psi.".format(relief1.read_close_pressure()))
```

```
    relief1.valve_operation(75)
    print(relief1.read_position())
    relief1.valve_operation(73)
    print(relief1.read_position())
```

The output of running `valve.py` on its own is shown in the following screenshot:


cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC
File Edit View Search Terminal Help
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC $ python3 ./Pip
ingSystems/valve/valve.py
Pump inlet created
Pump inlet is closed.
Pump inlet is open.
Pump inlet is closed.
Pump inlet is open.
Throttle valve created
Throttle valve is 0% open.
Throttle valve is 100% open.
Throttle valve is 0% open.
Throttle valve is 40% open.
Pressure relief created
Pressure relief is closed.
The open setpoint is 25 psi.
The close setpoint is 20 psi.
The open setpoint is 75 psi.
The close setpoint is 73 psi.
Pressure relief is open.
Pressure relief is closed.
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC $

Valve file output

Simulation pumps

The final main component of the fuel scenario are pumps. While the fuel simulation uses positive displacement pumps (a given amount of fluid is pumped out per cycle), we will also write code for centrifugal pumps (a type of variable displacement pump, which uses an impeller to impart velocity and pressure to the fluid) to make this program more universal.

Base pump class

First, we will define a base `Pump` class, much like we did for valves, as shown in the following code:

```
# pump.py (part 1)
1  #!/usr/bin/env python3
2  """
3  VirtualPLC pump.py
4
5  Purpose: Creates a generic Pump class for PLC-controlled SCADA systems.
6
7  Classes:
8      Pump: Generic superclass
9      CentrifPump: Pump subclass; provides for a variable displacement pump
10     PositiveDisplacement: Pump subclass; provides for a positive displacement pump
11
12  Author: Cody Jackson
13
14  Date: 4/12/18
15  ################################
16  Version 0.2
17          Added path extension to alleviate errors
18  Version 0.1
19      Initial build
20  """
```

Lines 1-20 are similar to what we've seen before, with the basic program information:

```
# pump.py (part 2)
1 import sys
2 sys.path.extend(["/home/cody/PycharmProjects/VirtualPLC"])
3 import math
4 import numbers
5 from Utilities import utility_formulas
6
7 GRAVITY = 9.81  # m/s^2
```

In lines 1 and 2, notice that, like the `Tank` program, we have to extend the system's path to include the root program directory to prevent errors when importing the utility formulas.

Line 7 defines the gravitational constant, which is used when calculating the power used by the pump. Unlike our other constants, this one is in metric values, as it is necessary for the pump power calculation:

```
# pump.py (part 3)
1  class Pump:
```

```
2       """Generic class for pumps.
3
4       Displacement is the amount of fluid pushed through the pump per second.
5       Horsepower coefficient is the slope of the equivalent pump curve.
6          """
7       def __init__(self, name="", flow_rate_out=0.0, pump_head_in=0.0, press_out=0.0,
pump_speed=0):
8            """Set initial parameters."""
9            self.name = name
10            self.__flow_rate_out = float(flow_rate_out)
11            self.head_in = float(pump_head_in)
12            self.__outlet_pressure = float(press_out)
13            self.__speed = pump_speed
14            self.__wattage = self.pump_power(self.__flow_rate_out,
self.diff_press_psi(self.head_in, self.outlet_pressure))
```

Line 1 starts the generic `Pump` class. The docstring (lines 2-6) provides some basic information about the class, while lines 7-14 initialize the pump instance parameters. For a pump, we care about the following:

- The pump instance's name
- The flow rate coming out of the pump
- The net positive suction head at the pump's inlet—this value has to be higher than the pump's minimum requirement to prevent cavitation
- The pump's outlet pressure
- The speed of the pump, typically measured in **revolutions per minute** (**rpm**)
- The power used by the pump, measured in watts

```
# pump.py (part 4)
1     @property
2     def speed(self):
3         """Get the current speed of the pump."""
4         return self.__speed
5
6     @speed.setter
7     def speed(self, new_speed):
8         """Change the pump speed."""
9         try:
10            if not isinstance(new_speed, numbers.Number):
11                raise TypeError("Numeric values only.")
12            elif new_speed < 0:
13                raise ValueError("Speed must be 0 or greater.")
14            else:
15                self.__speed = new_speed
16        except TypeError:
17            raise  # Re-raise error for testing
18        except ValueError:
19            raise  # Re-raise error for testing
```

The preceding code block defines the pump speed property and setter methods. Like the previous examples, name mangling is used for property variables to ensure there is no confusion with other names in the program. Notice that we are re-raising the exceptions in lines 17 and 19 for use when we write our unit tests:

```
# pump.py (part 5)
1     @property
2     def outlet_pressure(self):
3         """Get the current outlet pressure of the pump."""
4         return self.__outlet_pressure
5
```

```
6     @outlet_pressure.setter
7     def outlet_pressure(self, press):
8         """Set the pump outlet pressure."""
9         self.__outlet_pressure = press
10
11     @property
12     def flow(self):
13         """Get the current outlet flow rate of the pump."""
14         return self.__flow_rate_out
```

Lines 1-9 are the property and setter methods for the pump's outlet pressure, while lines 11-14 are for the output flow rate property method:

```
# pump.py (part 6)
1     @flow.setter
2     def flow(self, flow_rate):
3         """Set the pump outlet flow."""
4         self.__flow_rate_out = flow_rate
5
6     @property
7     def power(self):
8         """Get the current power draw of the pump."""
9         return self.__wattage
10
11     @power.setter
12     def power(self, power):
13         """Set the pump power."""
14         self.__wattage = power
```

Lines 1-4 are the outlet flow rate setter method and lines 6-14 are for the pump's power usage:

```
# pump.py (part 7)
1     def pump_power(self, flow_rate, diff_head, fluid_spec_weight=62.4):
2         """Calculate pump power in kW.
3
4         Formula from https://www.engineeringtoolbox.com/pumps-power-d_505.html.
5         Because imperial values are converted to metric, the calculation isn't
exactly the formula listed on the site; view the site's source code to see the formula
used.
6                 """
7         flow_rate = flow_rate / 15852
8         density = fluid_spec_weight / 0.0624
9         head = diff_head / 3.2808
10         hyd_power = (100 * (flow_rate * density * GRAVITY * head) / 1000) / 100
11         self.power = hyd_power
12         return self.power
```

The actual power calculation is handled by the function defined in the preceding code listing. As this formula was taken from a third-party site, testing the output was compared to the website's output to ensure correct values were used. This is important to note when using any external tools; the Python program's results have to be compared to the same tool, as rounding errors or other problems may skew the results:

```
# pump.py (part 8)
1     @staticmethod
2     def diff_press_ft(in_press_ft, out_press_ft):
3         """Calculate differential head across pump, converted from feet."""
4         in_press = utility_formulas.head_to_press(in_press_ft)
5         out_press = utility_formulas.head_to_press(out_press_ft)
6         delta_p = out_press - in_press
```

```
7         return delta_p
8
9     @staticmethod
10     def diff_press_psi(press_in, press_out):
11         """Calculate differential pump head."""
12         delta_p = abs(press_out - press_in)  # Account for Pout < Pin
13         return delta_p
```

The preceding code listing creates two static methods that calculate the pressure difference from the pump inlet to outlet; essentially, this is the increase in pressure from the pump. This method can be used by all instances of the class, as well as the class itself. Lines 1-7 return the pressure in feet of head pressure, while the static method in lines 9-13 returns PSI.

Centrifugal pump class

Having written all of the code for any generic pump, we now write the code for centrifugal pumps, which are the most common type of variable displacement pumps. This code is shown here:

```
# pump.py (part 9)
1class CentrifPump(Pump):
2    """Defines a variable-displacement, centrifugal-style pump."""
3
4    def get_speed_str(self):
5        """Get the current speed of the pump, in rpm."""
6        if self.speed == 0:
7            return "The pump is stopped."
8        else:
9            return "The pump is running at {speed} rpm.".format(speed=self.speed)
10
11     def get_flow_str(self):
12         """Get the current flow rate of the pump."""
13         return "The pump output flow rate is {flow} gpm.".format(flow=self.flow)
```

Line 1 subclasses the `Pump` base class to create the `CentrifPump` subclass. For this particular type of pump, we don't require any special parameters beyond the default, generic ones, so we don't have to provide a new `__init__` method.

Lines 4-9 create a method that returns a string, indicating the speed of the pump or if it is not running. Lines 11-13 do the same thing for pump flow rate:

```
# pump.py (part 10)
1    def get_press_str(self):
2        """Get the current output pressure for the pump."""
3        return "The pump pressure is {press:.2f}
psi.".format(press=self.outlet_pressure)
4
5    def get_power_str(self):
6        """Get the current power draw for the pump."""
7        return "The power usage for the pump is {pow:.2f} kW.".format(pow=self.power)
```

Lines 1-3 provide a string output for pump outlet pressure, and lines 5-7 are for pump power:

```
# pump.py (part 11)
1    def adjust_speed(self, new_speed):
2        """Defines pump characteristics that are based on pump speed.
3
4        Only applies to variable displacement (centrifugal) pumps. Variable names
match pump law equations.
5                """
6        n2 = new_speed  # Validate input
7
8        if self.speed == 0:  # Pump initially stopped
9            n1 = 1
10        else:
11            n1 = self.speed
12        v1 = self.flow
13        hp1 = self.outlet_pressure
14
15        self.flow = v1 * (n2 / n1)  # New flow rate
16        self.outlet_pressure = hp1 * math.pow((n2 / n1), 2)  # New outlet pressure
17        self.speed = n2  # Replace old speed with new value
18        delta_p = self.diff_press_psi(self.head_in,
utility_formulas.press_to_head(self.outlet_pressure))
19        self.power = self.pump_power(self.flow, delta_p)
```

The method that is defined in the following code listing adjusts the speed of the pump, but requires some explanation. Centrifugal pump operating characteristics are defined by the following pump affinity laws:

- Flow rate is proportional to pump speed ($Q \propto N$)
- Pressure (head) is proportional to the square of pump speed ($H \propto N^2$)
- Power is proportional to the cube of pump speed ($P \propto N^3$)

Hence, to double the flow rate from a centrifugal pump, the output pressure increases four-fold while the power required increases by a factor of eight.

The `adjust_speed()` method uses these pump laws to calculate the resulting changes as pump speed varies. It also attempts to account for a pump that is initially started, as the formulas assume a steady-state running condition:

```
# pump.py (part 12)
1    def start_pump(self, speed, flow, out_press=0.0, out_ft=0.0):
2        """System characteristics when a pump is initially started.
3
4        Assumes all valves fully open, i.e. maximum flow rate.
5
6        """
7        self.speed = speed
8        self.flow = flow
9        if out_press > 0.0:
10            self.outlet_pressure = out_press
11        elif out_ft > 0.0:
12            self.outlet_pressure = utility_formulas.head_to_press(out_ft)
13        else:
14            return "Outlet pump pressure required."
15        delta_p = self.outlet_pressure -
utility_formulas.head_to_press(self.head_in)
16        self.power = self.pump_power(self.flow, delta_p)
17
18        return self.speed, self.flow, self.outlet_pressure, self.power
```

The method in the preceding code listing initializes the pump's characteristics when it is initially started. This is only necessary when after a pump instance has been initially created and either the instance was set to a stopped condition or was shut off.

Positive displacement pump class

Positive displacement pumps are any pumps that have a set amount of fluid capacity per revolution, which includes rotary, reciprocating, and linear types. Examples include screw pumps, gear pumps, piston pumps, and diaphragm pumps.

The following code listings define the generic characteristics for any positive displacement pump; because it is a general class, any true simulation would require more specific engineering formulas:

```
# pump.py (part 13)
1  class PositiveDisplacement(Pump):
2      """Defines a positive-displacement pump."""
3
4      def __init__(self, name="", flow_rate_out=0.0, pump_head_in=0.0, press_out=0.0,
pump_speed=0, displacement=0.0):
5          super(PositiveDisplacement, self).__init__(name, flow_rate_out,
pump_head_in, press_out, pump_speed)
6          self.displacement = displacement
7
8      def get_speed_str(self):
9          """Get the current speed of the pump, in rpm."""
10          if self.speed == 0:
11              return "The pump is stopped."
12          else:
13              return "The pump is running at {speed} rpm.".format(speed=self.speed)
```

For this pump, we have to account for the fluid displacement that occurs per cycle of the pump, so we define a new `__init__` method in line 4 and create the new variable `displacement` in line 6.

Lines 8-13 is a string return method for speed, similar to the centrifugal pump class:

```
# pump.py (part 14)
1    def get_flow_str(self):
2        """Get the current flow rate of the pump."""
3        return "The pump outlet flow rate is {flow} gpm.".format(flow=self.flow)
4
5    def get_press_str(self):
6        """Get the current output pressure for the pump."""
7        return "The pump pressure is {press:.2f}
psi.".format(press=self.outlet_pressure)
8
9    def get_power_str(self):
10        """Get the current power draw for the pump."""
11        return "The power usage for the pump is {pow:.2f}
kW.".format(pow=self.power)
```

The methods listed here are the same types of methods as the centrifugal pump and require no additional explanation.

```
#  pump.py (part 15)
1 def adjust_speed(self, new_speed):
2         """Modify the speed of the pump, assuming constant outlet pressure.
3
4         Affects the outlet flow rate and power requirements for the pump.
5         """
6        self.speed = new_speed
7        press_in = utility_formulas.head_to_press(self.head_in)
8
9       self.flow = self.speed * self.displacement
10       self.power = self.pump_power(self.flow, self.diff_press_psi(press_in,
self.outlet_pressure))
11 # TODO: Account for different flow rates based on outlet pressure
```

The speed of a positive displacement pump changes the amount of fluid pumped per cycle, as a set amount is moved each time. Therefore, a new speed method is created in the preceding code listing.

Line 11 is a TODO reminder that we will have to return later and figure out how to calculate changes in flow rate based on system pressure. Positive displacement pumps function differently than centrifugal pumps when it comes to outlet pressure. While centrifugal pumps can operate even if the outlet valve is completely shut, positive displacement pumps will continue to displace the same amount of fluid with each revolution, regardless of any obstructions. If it continues long enough, the piping or connections will break due to overpressure.

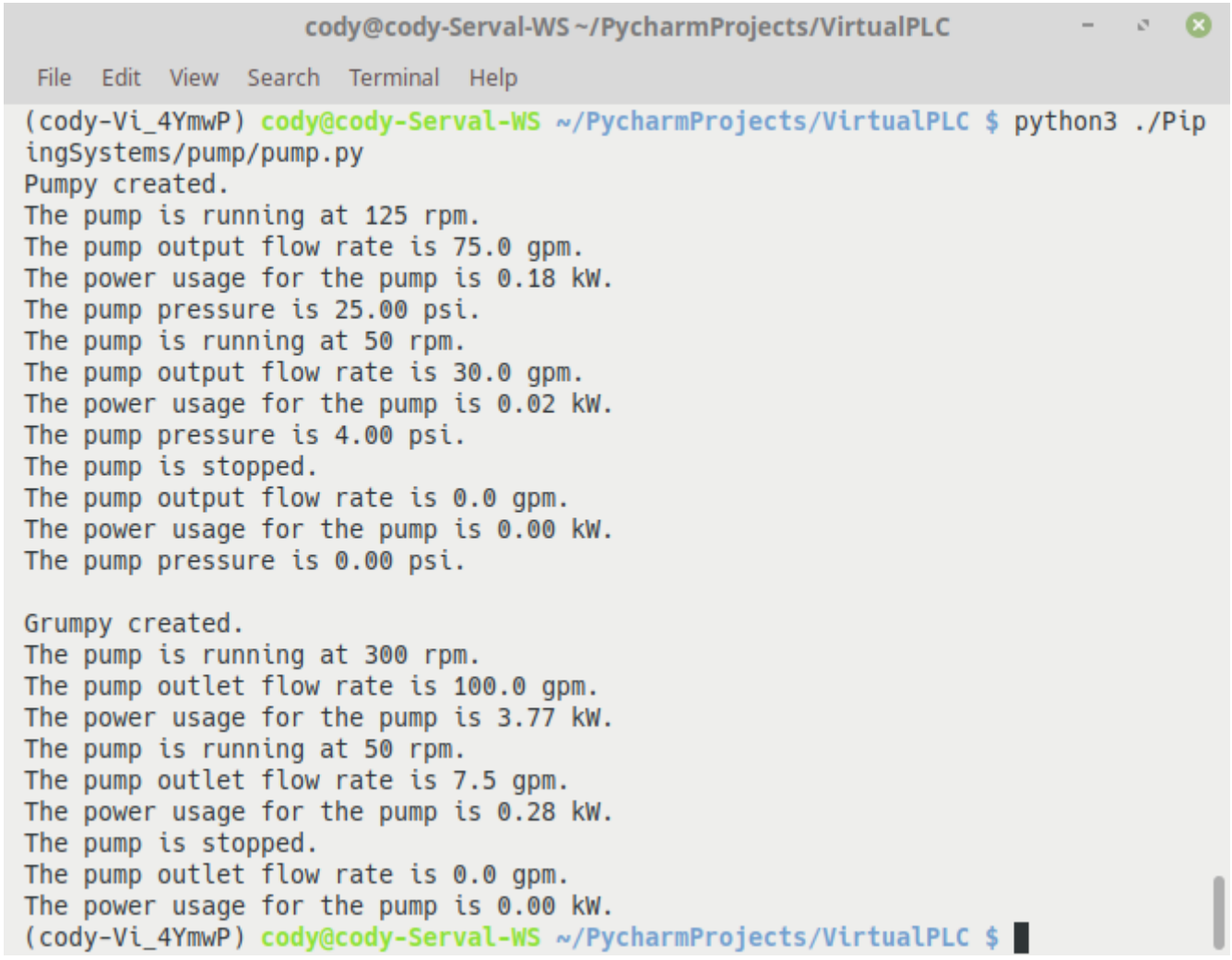
In other words, with a given speed, a positive displacement pump will attempt to maintain a constant flow rate. The actual flow rate will change based on the backpressure seen on the outlet of the pump, which is shown in the manufacturer's documentation. Currently, this pump program does not account for these real-world changes, but should be included in the future to provide a better simulation.

The basic functionality tests for the pump program are shown next, in two different code listings:

```
# pump.py (part 16)
1 if __name__ == "__main__":
2     # Functional test_valves
3     # name="", flow_rate=0.0, pump_head_in=0.0, press_out=0.0, pump_speed=0, hp=0.0,
displacement=0.0
4     pump1 = CentrifPump("Pumpy", 75, 12, 25, 125)
5     print("{} created.".format(pump1.name))
6     print(pump1.get_speed_str())
7     print(pump1.get_flow_str())
8     print(pump1.get_power_str())
9     print(pump1.get_press_str())
10     pump1.adjust_speed(50)
11     print(pump1.get_speed_str())
12     print(pump1.get_flow_str())
13     print(pump1.get_power_str())
14     print(pump1.get_press_str())
15     pump1.adjust_speed(0)
16     print(pump1.get_speed_str())
17     print(pump1.get_flow_str())
18     print(pump1.get_power_str())
19     print(pump1.get_press_str())
# pump.py (part 17)
1    pump2 = PositiveDisplacement("Grumpy", 100, 0, 200, 300, 0.15)
2    print("\n{} created.".format(pump2.name))
3    print(pump2.get_speed_str())
4    print(pump2.get_flow_str())
5    print(pump2.get_power_str())
6    pump2.adjust_speed(50)
```

```
7     print(pump2.get_speed_str())
8     print(pump2.get_flow_str())
9     print(pump2.get_power_str())
10     pump2.adjust_speed(0)
11     print(pump2.get_speed_str())
12     print(pump2.get_flow_str())
13     print(pump2.get_power_str())
```

The output of these tests are shown in following screenshot:


cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC
File Edit View Search Terminal Help
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC $ python3 ./Pip
ingSystems/pump/pump.py
Pumpy created.
The pump is running at 125 rpm.
The pump output flow rate is 75.0 gpm.
The power usage for the pump is 0.18 kW.
The pump pressure is 25.00 psi.
The pump is running at 50 rpm.
The pump output flow rate is 30.0 gpm.
The power usage for the pump is 0.02 kW.
The pump pressure is 4.00 psi.
The pump is stopped.
The pump output flow rate is 0.0 gpm.
The power usage for the pump is 0.00 kW.
The pump pressure is 0.00 psi.
Grumpy created.
The pump is running at 300 rpm.
The pump outlet flow rate is 100.0 gpm.
The power usage for the pump is 3.77 kW.
The pump is running at 50 rpm.
The pump outlet flow rate is 7.5 gpm.
The power usage for the pump is 0.28 kW.
The pump is stopped.
The pump outlet flow rate is 0.0 gpm.
The power usage for the pump is 0.00 kW.
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC $

Pump test output

Summary

In this chapter, we developed the requirements for a generic liquid storage and transfer system, for later use in a fueling scenario. We created several utility functions that will help with individual component calculations, as well as writing the code for storage tanks, valves, and pumps.

In the next chapter, we will learn how to write automated tests to double-check our code, as well as alerting us if code changes cause breakages.

Automated Software Testing

ISoftware testing is performed to ensure the final product will meet the needs of the customer or stakeholders. Software is checked to see whether all of the expected output is generated and that no overly detrimental errors occur. No program is bug-free, and enhancements are often made, so testing needs to be a firm part of the product life cycle.

To be honest, I didn't learn about writing tests until fairly recently. After listening to a Python podcast, this appears to be a common problem with developers; some computer science degrees never teach testing within the curriculum, so a large number of programmers are forced to learn on their own.

After writing small programs to gain familiarity with testing, the fuel farm scenario in this book was the first large-scale implementation of software testing I performed, as the scenario was to be used in a training program. Therefore, I had to make sure that the code worked as expected and the physical parameters were correct.

In this chapter, we will cover the following topics:

- Testing techniques
- Writing tests
- Refactoring code

Testing techniques

When testing software, a wide range of philosophies and methodologies have been developed over the years. Some people write the entire program and then attempt to manually check every user action to verify the results. Others write their tests first (knowing they will fail), then write the code to make the tests pass. We will discuss some of the more common testing methods in this section.

Static versus dynamic tests

Static testing generally comprises a review of the code without trying to run it. This includes actually looking through the code, an action similar to proofreading, as well as using tools such as text editors or compilers to check syntax and logic flow.

Dynamic tests run the program, or portions of it, to look at the inner workings (as is the case when using a debugger) or just to validate the expected output.

White-box testing

White-box testing checks the internal workings of the software rather than the user experience. It looks at the source code and follows different paths through the code, such as all of the possible branches of `if...else` statements.

In addition, white-box testing considers all the possible connections between the different parts of an application. So, not only will a particular unit, such as a class, be looked at, but also all of the classes within a module and all of the modules within a program.

The problem with white-box testing is that it can only identify problems in code that has been implemented. If a particular condition is marked as **TODO**, but no logic has been written to even call the condition, such as a simple code stub, then testing won't be able to identify that a project requirement is missing.

White-box testing can include the following:

- **Application Programming Interface** (**API**) **testing**: This means checking public and private APIs. An API is simply a way to interact with the program without having to know the inner workings. For example, in the preceding fuel program, using the `close()` method changes the position of a valve and ensures that all of the outlet flow and pressure values are set to zero. The method allows a programmer to work with the program without requiring knowledge of how the model works.
- **Code coverage**: This means creating tests that check the desired amount of code. It may not be possible, or necessary, to check 100% of an application, but some coding shops require a certain amount of code to be checked, whether it is a percentage of the whole code base or a certain number of classes, functions, and so on.
- **Fault injection**: This means intentionally feeding bad data or other faults into the software and checking the results. In security testing, fuzzing performs a similar function, where a variety of data is fed into a system to see what potential vulnerabilities exist.
- **Mutation testing**: This means checking the quality of software tests themselves by checking how effective they are in preventing changes to the code. The program is modified slightly (mutated) and the tests are run. The tests look for changes in the code and, hopefully, reject the changes. The percentage of mutants that are **killed** (caught by tests and rejected) is the rating of the test suite. Mutants often mimic programming errors, such as incorrect variables, or logic errors, such as dividing by zero.
- **Static testing**: This means looking at the code itself without actually running it.

Black-box testing

Black-box testing could be considered more of a quality-control/quality-assurance check. The application is tested based on what a user would expect to deal with, without any knowledge of the source code itself; the software is considered a "black box", and only the input and output is known, not how it is manipulated.

Some of black-box techniques include the following :

- **Equivalence partitioning**: Input is partitioned into equivalent data from which tests are created. Ideally, each partition is tested at least once. The purpose is to identify classes of errors rather than individual errors, thereby reducing the total number of tests that have to be written.
- **Boundary-value analysis**: Within a set of values, the minimum and maximum values are tested, allowing easy elimination of everything in between as potential errors.
- **All-pairs testing**: If all possible input combinations are identified, then testing all possible, discrete pairings can help speed parameter testing.
- **State testing**: Using one of several different "state machines" (such as a state table or state diagram), all possible states of a system are identified and tested. Input includes normal input parameters, as well as the current state, while output includes normal output parameters and the new state.
- **Decision table**: A visual representation of `if...else` statements, `switch` cases, and so on. An example of this is a troubleshooting guide in a vendor manual. A decision table can be used to validate the various conditions in order to verify whether everything is accounted for in both expected and unexpected use cases. As a table, it could also be made into a database or another computer-readable format to provide automated testing.
- **Fuzzing**: This uses automated software to produce a variety of input values to determine how they affect system outcome, such as crashes, error generation, and memory leaks.
- **Use cases**: A list of actions that defines a variety of expected uses of the software and the expected outcomes. This is often a key part of agile programming, as it helps direct a programming team in the final product's capabilities.

When to test

Tests can be written before, during, or after coding the project. **Test-Driven Development**(**TDD**) uses the idea of short development cycles, with each cycle based on a set number of features to implement. Prior to coding, tests are written (and expected to fail) that dictate the end states of the different features. The actual code is written with an eye to making the tests pass; once a test passes, no further work is necessary for that particular feature.

The problem with TDD is that it focuses on small portions of the overall project, but doesn't necessarily address full functional testing of the product. It also means that management has to support the TDD paradigm. Many managers feel testing is a waste of time or, at best, left until the end of the project; they often feel that writing tests doesn't add to meeting deadlines because it isn't functional code.

TDD also means that the tests need to be thought out and designed to address the needs of the project. Since the main code is written to make tests pass, it's entirely possible that portions of the project aren't written because someone forgot to write the tests for it.

More common, at least in this my experience, is testing after development. While a variety of agile programming techniques dictate how the project code is written, the tests are written after most of the project is completed.

The advantage is that all aspects of the project can be tested, because the developers already known what is present. A disadvantage is that some of the project code could be wasteful if it was written without a purpose; in other words, while TDD allows the programmer to stop writing code when a test passes, writing tests after the fact means that more tests may have to be written, and some project code may have been written that wasn't necessary to meet project requirements.

Ultimately, while it is better to test early and test often, having tests at all is more important than not having any tests.

Writing tests

Python has a number of different testing libraries available. The default library that comes with Python is `unittest`. This library is based on JUnit, from Java, and it shows. In this my opinion, the library is not especially user friendly, and there are 12 different tests to choose from, based on the expected outcome of the code. Writing the tests isn't especially intuitive for beginners, partly because of the amount of boilerplate code required just to work with `unittest`.

While `nose2` is available as a third-party testing library, a more popular option is `pytest`. It requires no boilerplate; most of the time, just having `pytest` installed on your system is sufficient, though, in some cases, an explicit import of `pytest` is required. Tests are written as you would write normal Python code; the `assert` keyword tells the testing framework what the expected outcome is.

The easiest way to learn how to test is to see the code, which is especially useful in this case since `pytest` is so easy to use. The test file that we will cover in this section is separated into different sections for clarity, but they are all part of the same file.

It should be noted that, for `pytest` to work properly, all test files have to start with the word `test_`. In addition, the following test file contains arbitrary components; it is simply testing whether the class methods we have written perform as expected, prior to writing the actual simulation:

```
# test_functions.py (part 1)
1  """Assumes valves in series, with the first supplied by a tank 10 feet above the
valve with a pipe length of 6 feet.
2  Water level is 4 feet above tank bottom; total water head = 14 feet.
3  """
4  from PipingSystems.pump.pump import CentrifPump, PositiveDisplacement
5  from PipingSystems.valve.valve import Gate, Globe, Relief
6
7  valve1 = Gate("Valve 1", position=100, flow_coeff=200,
sys_flow_in=utility_formulas.gravity_flow_rate(2, 1.67),
press_in=utility_formulas.static_press(14))
8  pump1 = CentrifPump("Pump 1")
9  throttle1 = Globe("Throttle 1", position=100, flow_coeff=21)
10 valve2 = Gate("Valve 2", position=100, flow_coeff=200)
11 valve3 = Gate("Valve 3", position=100, flow_coeff=200)
```

Lines 1-3 provide some basic background assumptions for this test file. This helps when double-checking the results.

We have to import the items that we want to test, so the specific components are identified in lines 4 and 5.

Starting with line 7, we create instances of each component and provide the initial values. A position of 100 means that the valve is fully open. The flow coefficient represents how much the valve's constructions affects the flow that passes through it; a higher value indicates a lesser impact on the flow rate and pressure drop. Look at the following code:

```
# test_functions.py (part 2)
1  pump2 = PositiveDisplacement("Gear Pump", displacement=0.096, press_out=30)
2  relief1 = Relief("Relief 1", position=0, open_press=60, close_press=55)
3  recirc1 = Globe("Throttle 2", position=100, flow_coeff=21)
4  valve4 = Gate("Valve 4", position=100, flow_coeff=200)
5
6  # Utility functions
7  def test_grav_flow():
8      flow_rate = utility_formulas.gravity_flow_rate(2, 1.67)
9      assert flow_rate == 319.28008077388426
10
11
12 def test_static_press():
13     press = utility_formulas.static_press(14)
14     assert press == 6.068373888888889
```

Lines 1-4 continue the creation of the component instances we will test. Starting with line 8, we define the functions that will test our utility formulas.

When using `pytest`, we define the test function (starting with the word `test_`), followed by the normal code logic that we expect to use in the final product (lines 8 and 13). Once all of the functionality has been defined, we create one or more `assert` statements that test the final outcome against its expected value (lines 9 and 15).

Depending on how you want to write your tests, you can accept the default precision of Python calculations, as shown in the following code, or you can truncate/round the results to the desired precision. A case could be made either way, as it doesn't require any extra effort to use the default, but if your final value is off by one value, the entire assertion errors out:

```
# test_functions.py (part 3)
class TestSystem:

# Gate Valve 1
def test_v1_press_in(self):
    assert valve1.press_in == 6.068373888888889

def test_v1_flow_in(self):
    assert valve1.flow_in == 319.28008077388426

def test_v1_flow_out(self):
    valve1.flow_out = valve1.flow_in
    assert valve1.flow_out == 319.28008077388426
```

In the preceding code listing, we can see an alternative way to use `pytest`. You can use individual functions, as demonstrated in part 2, or you can make a test class and define methods for each test case. This is useful if you have multiple tests that relate to each other. In this example, we are simply testing a single system with limited components; you could also have a separate class for each component type, different subsystems, and so on.

For this test suite, we will test the input/output flow rates and pressure values for the components, based on our expectations, as shown in the following code:

```
# test_functions.py (part 4)
def test_v1_press_drop(self):
    valve1.press_drop(valve1.flow_out)
    assert valve1.deltaP == 2.5484942494744516

def test_v1_press_out(self):
    valve1.get_press_out(valve1.press_in)
    assert valve1.press_out == 3.5198796394144374

# Centrifugal Pump
def test_pump1_input_press(self):
    pump1.head_in = utility_formulas.press_to_head(valve1.press_out)
    assert pump1.head_in == 8.119222584669064
```

Following code snippet is the part 5 of `test_functions.py`:

```
# test_functions.py (part 5)
def test_pump1_start_pump(self):
    pump1.start_pump(1750, 50, 16)
    assert pump1.speed == 1750
    assert pump1.flow == 50.0
    assert pump1.outlet_pressure == 16
    assert pump1.power == 0.11770474358069433

# Globe valve 1
def test_t1_press_in(self):
    throttle1.press_in = pump1.outlet_pressure
    assert throttle1.press_in == 16
```

In the preceding code listing, we can see that multiple `assert` statements can be used within a single test case. Here, once a pump is started, we want to ensure that all of the pump parameters are set correctly. Rather than writing a separate test case for each condition, we can put all assertions into one case, as shown in the previous code. The following code snippet is part 6 of `test_functions.py`

```
# test_functions.py (part 6)
def test_t1_flow_in(self):
throttle1.flow_in = pump1.flow
assert throttle1.flow_in == 50.0

def test_t1_flow_out(self):
throttle1.flow_out = throttle1.flow_in
assert throttle1.flow_out == 50.0

def test_t1_press_drop(self):
throttle1.press_drop(throttle1.flow_out)
assert throttle1.deltaP == 5.668934240362812

def test_t1_press_out(self):
throttle1.get_press_out(throttle1.press_in)
assert throttle1.press_out == 10.331065759637188
```

Following code snippet is the part 7 of `test_functions.py`:

```
# test_functions.py (part 7)
# Gate Valve 2
def test_v2_input_press(self):
    valve2.press_in = throttle1.press_out
    assert valve2.press_in == 10.331065759637188

def test_v2_input_flow(self):
    valve2.flow_in = throttle1.flow_out
```

```
    assert valve2.flow_in == 50.0


def test_v2_output_flow(self):
    valve2.flow_out = valve2.flow_in
    assert valve2.flow_out == 50.0
```

Following code snippet is the part 8 of `test_functions.py`:

```
# test_functions.py (part 8)
def test_v2_press_drop(self):
    valve2.press_drop(valve2.flow_out)
    assert valve2.deltaP == 0.0625


def test_v2_press_out(self):
    valve2.get_press_out(valve2.press_in)
    assert valve2.press_out == 10.268565759637188

# Gate Valve 3
def test_v3_input_press(self):
    valve3.press_in = valve2.press_out
    assert valve3.press_in == 10.268565759637188
```

Following code snippet is the part 9 of `test_functions.py`:

```
# test_functions.py (part 9)
def test_v3_input_flow(self):
    valve3.flow_in = valve2.flow_out
    assert valve3.flow_in == 50.0

def test_v3_output_flow(self):
    valve3.flow_out = valve3.flow_in
    assert valve3.flow_out == 50.0

def test_v3_press_drop(self):
    valve3.press_drop(valve3.flow_out)
    assert valve3.deltaP == 0.0625

def test_v3_press_out(self):
    valve3.get_press_out(valve3.press_in)
    assert valve3.press_out == 10.206065759637188
```

Following code snippet is the part 10 of `test_functions.py`:

```
# test_functions.py (part 10)
# Gear Pump
def test_pump2_input_press(self):
    pump2.head_in = utility_formulas.press_to_head(valve3.press_out)
    assert pump2.head_in == 23.542088964737797

def test_pump2_output(self):
    pump2.adjust_speed(300)
    assert pump2.speed == 300
    assert pump2.flow == 28.8
    assert pump2.power == 0.10753003776038036

# Relief Valve 1
def test_relief1_input_press(self):
    relief1.press_in = pump2.outlet_pressure
    assert relief1.press_in == 30
```

Following code snippet is the part 11 of `test_functions.py`:

```
# test_functions.py (part 11)
# Globe Valve 2
def test_t2_input_press(self):
    recirc1.press_in = pump2.outlet_pressure
    assert recirc1.press_in == 30

def test_t2_input_flow(self):
    recirc1.flow_in = pump2.flow
    assert recirc1.flow_in == 28.8

def test_t2_output_flow(self):
    recirc1.flow_out = recirc1.flow_in
    assert recirc1.flow_out == 28.8

def test_2_press_drop(self):
    recirc1.press_drop(recirc1.flow_out)
    assert recirc1.deltaP == 1.8808163265306124
```

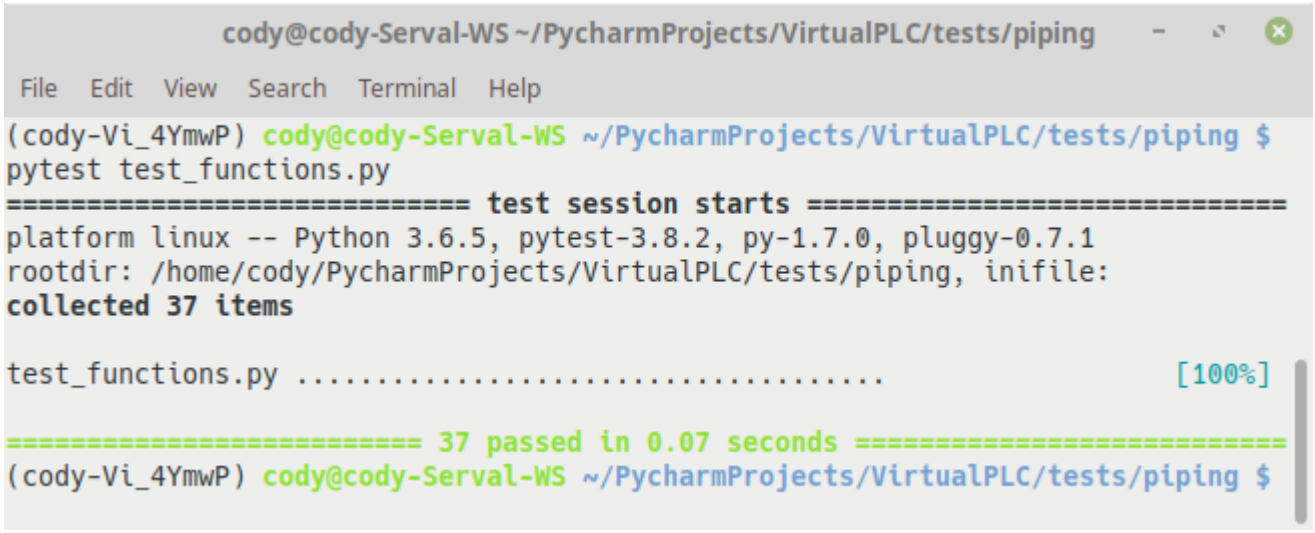
Following code snippet is the part 12 of `test_functions.py`:

```
# test_functions.py (part 12)
def test_t2_press_out(self):
    recirc1.get_press_out(recirc1.press_in)
    assert recirc1.press_out == 28.119183673469387

# Gate Valve 4
def test_v4_input_press(self):
    valve4.press_in = recirc1.press_out
    assert valve4.press_in == 28.119183673469387

def test_v4_input_flow(self):
    valve4.flow_in = recirc1.flow_out
    assert valve4.flow_in == 28.8
```

Following code snippet is the part 13 of `test_functions.py`:

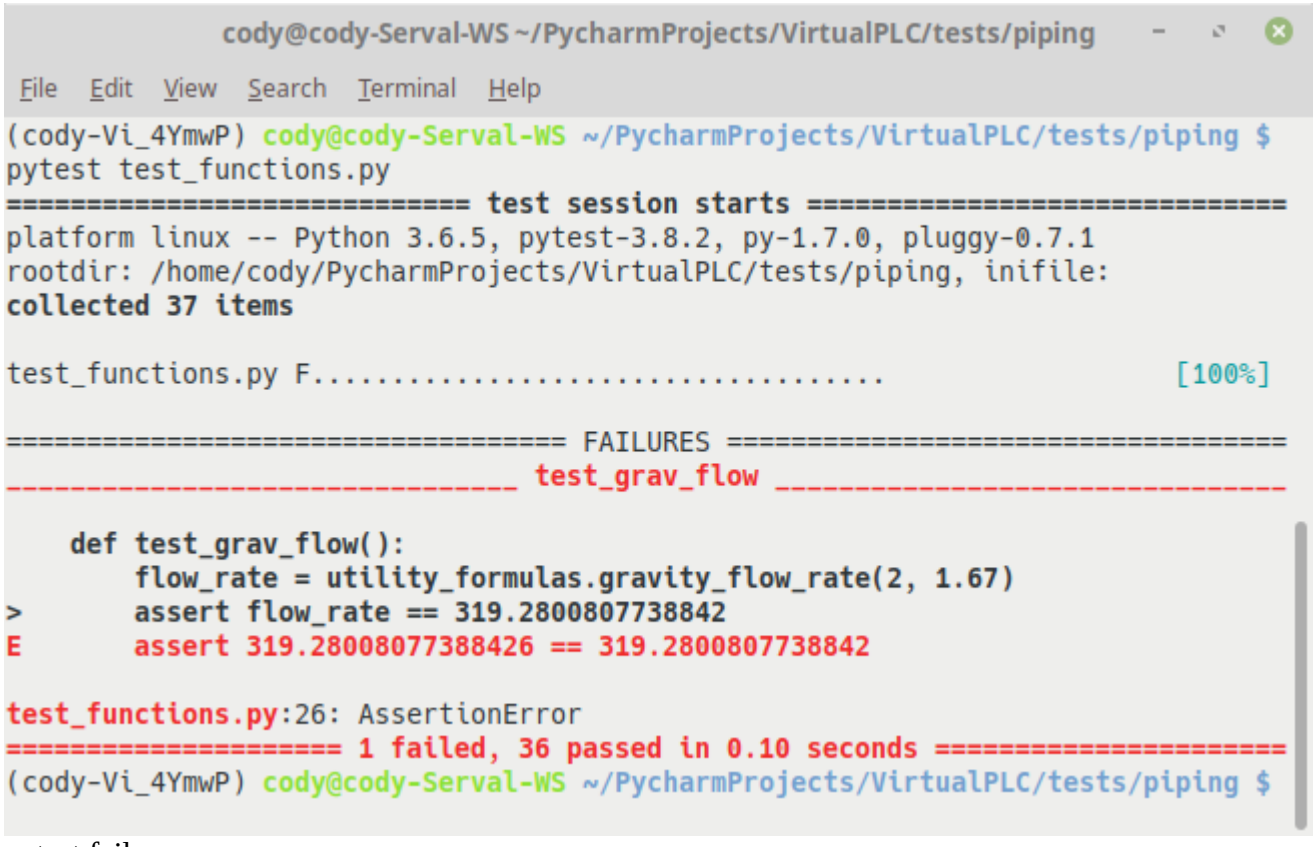
```
# test_functions.py (part 13)
def test_v4_output_flow(self):
    valve4.flow_out = valve4.flow_in
    assert valve4.flow_out == 28.8

def test_v4_press_drop(self):
    valve4.press_drop(valve4.flow_out)
    assert valve4.deltaP == 0.020736000000000004

def test_v4_press_out(self):
    valve4.get_press_out(valve4.press_in)
    assert valve4.press_out == 28.098447673469387
```

Assuming that we have written the tests correctly, this will be completed successfully. The following screenshot shows successful completion:


cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests/piping
File Edit View Search Terminal Help
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests/piping $
pytest test_functions.py
============================= test session starts ==============================
platform linux -- Python 3.6.5, pytest-3.8.2, py-1.7.0, pluggy-0.7.1
rootdir: /home/cody/PycharmProjects/VirtualPLC/tests/piping, inifile:
collected 37 items
test_functions.py .... [100%]
========================== 37 passed in 0.07 seconds ===========================
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests/piping $
A pytest success

Each of the dots represents a test case that has successfully passed. We also receive information on the total number of tests that were processed, as well as the total time required to perform all of the tests.

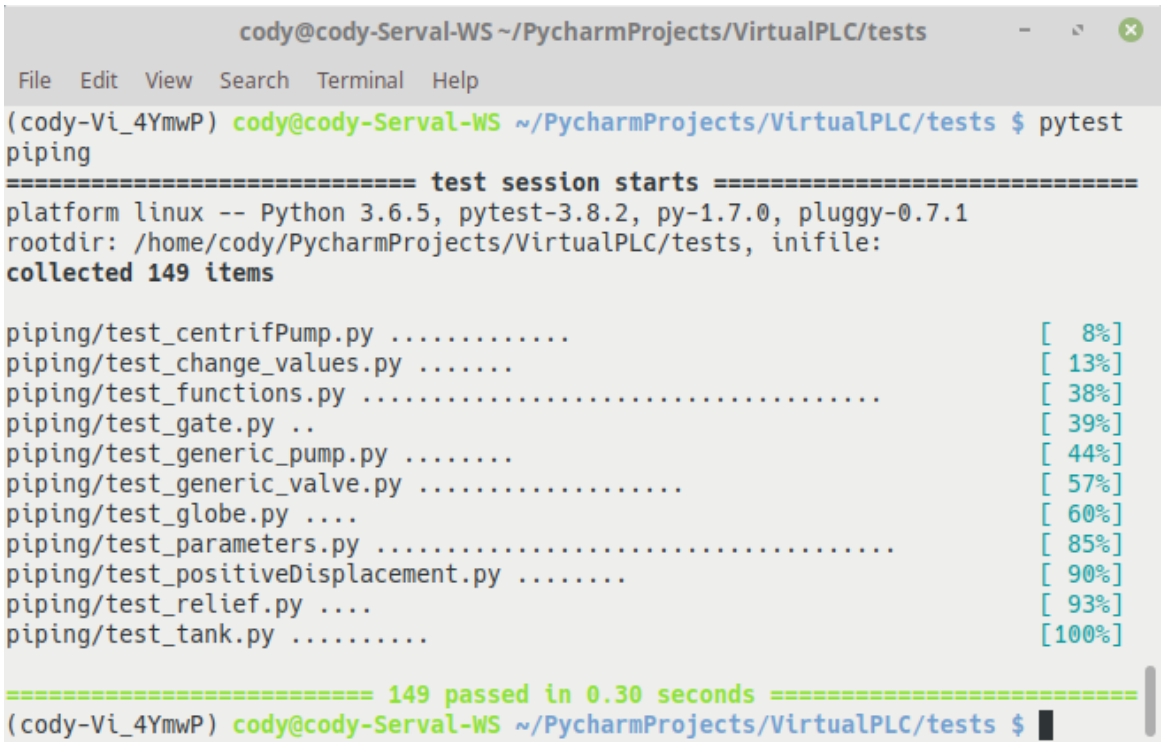
If we had a problem, we would see something like the following screenshot:


cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests/piping
File Edit View Search Terminal Help
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests/piping $
pytest test_functions.py
============================= test session starts ==============================
platform linux -- Python 3.6.5, pytest-3.8.2, py-1.7.0, pluggy-0.7.1
rootdir: /home/cody/PycharmProjects/VirtualPLC/tests/piping, inifile:
collected 37 items
test_functions.py F..................................... [100%]
================================== FAILURES ===================================
_____________________________ test_grav_flow ______________________________
def test_grav_flow():
flow_rate = utility_formulas.gravity_flow_rate(2, 1.67)
> assert flow_rate == 319.2800807738842
E assert 319.28008077388426 == 319.2800807738842
test_functions.py:26: AssertionError
==================== 1 failed, 36 passed in 0.10 seconds =====================
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests/piping $
pytest failure

The `pytest` library is helpful in telling us the problem. First, where we saw a row of dots previously, a capital F shows where the failure occurred among all of the test cases.

Second, a FAILURES section provides the details. In this instance, it tells us the test case that failed (`test_grav_flow()`), the specific assertion statement that failed, and what the actual received result was compared to the expected value. The expected value is what we have written into the test case; in this case, we deleted the final number from the calculated flow rate, resulting in the error.

There are many other tests that can be written, such as testing a generic pump instance, and testing a specific gate valve instance. The following screenshot shows how to run multiple test files at one time (these test files are provided in the code repository for this book):


cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests
File Edit View Search Terminal Help
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests $ pytest
piping
============================= test session starts ==============================
platform linux -- Python 3.6.5, pytest-3.8.2, py-1.7.0, pluggy-0.7.1
rootdir: /home/cody/PycharmProjects/VirtualPLC/tests, inifile:
collected 149 items
piping/test_centrifPump.py ..... [8%]
piping/test_change_values.py ..... [13%]
piping/test_functions.py ..... [38%]
piping/test_gate.py .. [39%]
piping/test_generic_pump.py ..... [44%]
piping/test_generic_valve.py ..... [57%]
piping/test_globe.py .... [60%]
piping/test_parameters.py ..... [85%]
piping/test_positiveDisplacement.py ..... [90%]
piping/test_relief.py .... [93%]
piping/test_tank.py ..... [100%]
========================== 149 passed in 0.30 seconds ==========================
(cody-Vi_4YmwP) cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC/tests $
Multiple pytest cases

With the preceding test run, we pointed `pytest` at a directory that contained the test files we wanted to run. As shown in the test results, 149 test cases were found. The `pytest` library will process all of the tests within each file, providing a running count of the total percentage completed after each file.

One final thing to point out about tests refers back to re-raising exceptions in our code. When an exception is re-raised after catching it, it allows a test case to capture the exception and run an `assert` statement against it, just like the calculations we've tested. The following code listing provides an example of this (this is just one test method from the entire test file):

```
def test_tank_level_str(self):
    tank1 = Tank()
    with pytest.raises(TypeError) as excinfo:
        tank1.level = "a"
    exception_msg = excinfo.value.args[0]
    assert exception_msg == "Numeric values only."
```

In this test, we want to check whether an exception is raised if a non-numeric value is provided as an argument to the tank-level method in `tank.py`. Doing this allows us to ensure that the correct exception is generated, as well as the error message that is printed, thereby allowing multiple exceptions with different messages to be tested.

Refactoring code

When debugging, testing, and troubleshooting, code is often rewritten to change how it works, a process that is also known as refactoring. The benefit of having test cases available means that changes to the code can be checked to see whether functionality has been compromised, such as interactions between functions or methods or if end-user complications have developed.

In large testing projects, continuous integration test suites are utilized to check code as the developers upload it to the server. These tests not only check each developer's code, but also check it against other code that was submitted to ensure that the entire project continues to function properly.

Refactoring can take many different forms, such as the following:

- Adding comments to help clarify key bits of code or why a certain piece of logic was used.
- Improving readability by separating code into logical blocks.
- Abstracting code into more general types. For example, if we had started the fuel scenario by writing the code for a globe valve, we could refactor it by abstracting the valve parameters to make a generic valve class.
- Encapsulation fields, which force code to use getter/setter methods rather than direct access.
- Replacing conditional behavior with class polymorphism. In other words, instead of testing for different conditions that dictate different logic paths, we create a class with a method that can account for the different conditions.
- Separating functions/methods that perform several jobs into multiple ones, where each one does only one job.
- Moving methods to more appropriate classes or even modules.
- Renaming objects to make them readily understandable.
- Converting a class into a superclass (pull up) and converting other classes into subclasses (push down).

When using an IDE, a number of refactoring tools may be available with just a click of a mouse. For example, the PyCharm IDE provides the following refactoring tools:

- **Rename**: This renames anything, from a variable or function to modules and projects.
- **Change signature**: This can be used to rename functions, add/remove parameters, assign default values, or reorder parameters.
- **Move**: This moves an object to another location within the code directory.
- **Copy**: This copies an object to another location within the code directory.
- **Safe delete**: Before deletion, PyCharm will check for locations where the file is being called. If found, the user will be allowed to make changes to the code prior to deletion.
- **Extract**: This addresses expressions that are hard to understand or are duplicated in the code by placing the expression result into a separate variable that is less complex and, therefore, easier to understand.
- **Parameter**: This adds a new parameter to a function declaration and automatically updates function calls.
- **Superclass**: This creates a superclass from the existing class or renames the existing class to become an implementation for a new superclass.
- **Constant**: This allows the conversion of multiple occurrences of an expression into one constant.
- **Field**: This allows the declaration of a new field and its initialization with a selected expression.
- **Method**: This takes a code fragment that can be grouped together and creates a separate method with it.
- **Variable**: This allows conversion of a duplicated expression, or one that is hard to understand, into a separate variable that is less complex.
- **Inline**: The following items are the opposite implementation of their respective "extract" tools:
    - Variable

    - Constant
    - Field
    - Parameter
    - Method
    - Superclass
- **Invert Boolean**: This changes any Boolean value to its opposite. This also changes its respective usage.
- **Pull members up**: This moves class members to a superclass.
- **Push members down**: This moves class members to a subclass.
- **Convert in Python package**: This automatically converts a directory or subdirectory into a Python package with the required `__init__.py` file.
- **Convert in Python module**: This consolidate all modules from a package into a single module.

Summary

In this chapter, we learned about why testing is important, the different methodologies regarding software testing, and how to write tests using `pytest`, and we looked at some concepts and tools to use when refactoring code.

In the next chapter, we will take our simulated components and actually write the code to simulate a portion of the schematic drawing from `Chapter 7`, *Writing the Imported Program*, the diagram entitled *Project schematic diagram*. We will also write some tests to verify that the functionality of the simulation works correctly.

Writing the Fueling Scenario

In the last two chapters, we learned about writing the code that creates the components for our final project, as well as writing tests to confirm that the logic works correctly. That actually doesn't do anything for us besides laying the groundwork for creating different piping and fluid-transfer scenarios.

In this chapter, we will plan how the final scenario (a fuel-storage and fuel-transfer system), based on the schematic drawing, will actually function, write the code that creates the components and their associated functionality, and, finally, write tests to ensure that the coded design actually works.

In this chapter, we will cover the following topics:

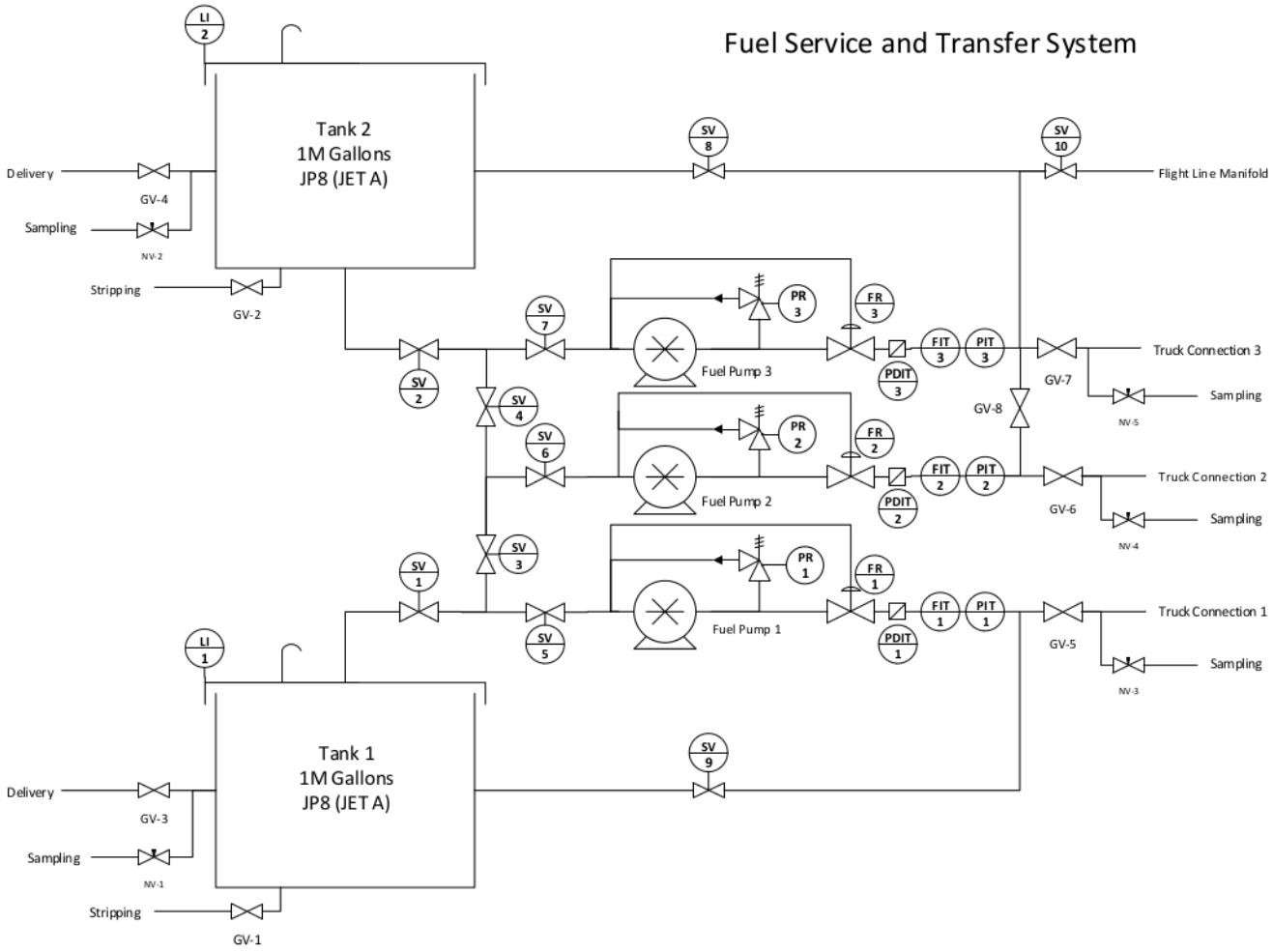
- Fueling scenario requirements
- Directory structure
- Component coding
- Functionality coding
- Testing

Fueling scenario requirements

We established the generic fluid simulation requirements in `Chapter 7`, *Writing the Imported Program*. Now we will determine the requirements for creating a fuel farm. A fuel farm is a storage and transfer location for fuel, often seen at airports, trucking companies, and so on.

The fuel tanks can be above or below ground, and often comprise both large storage tanks (holding millions of gallons of fuel) and service tanks (holding hundreds of gallons of fuel). Storage tanks are just that, storage for fuel prior to its transfer to the service tanks. Service tanks hold the fuel that is pumped into a vehicle, or otherwise put to use. Typically, there is equipment to separate water and particulate matter from the fuel as it is moved from the storage tank to the service tank. In addition, sample lines allow fuel to be tested for a number of factors such as color, clarity, and so on.

Our project is based on the schematic drawing that we saw before in `Chapter 7`, *Writing the Imported Program,* and which is reprinted in the following image:



Fuel farm schematic

In this drawing, there are no service tanks, so the fuel is taken directly from the storage tanks, through the pumps, and out to either the airport flight line or a fuel truck. In our code, we will only model the following components:

- Pumps 1-3
- Gate valves 1-10
- Pressure relief valves 1-3
- Outlet pressure regulating valves 1-3
- Storage tanks 1 and 2

The reason we will only model these components is that these have the largest effect on the system. After the pressure regulating valves, it doesn't really matter what the valves are doing, as the regulating valves will ensure that the outlet pressure of the pumps remains constant. In addition, these devices can be remotely operated, which will be a key component when we make a graphical interface.

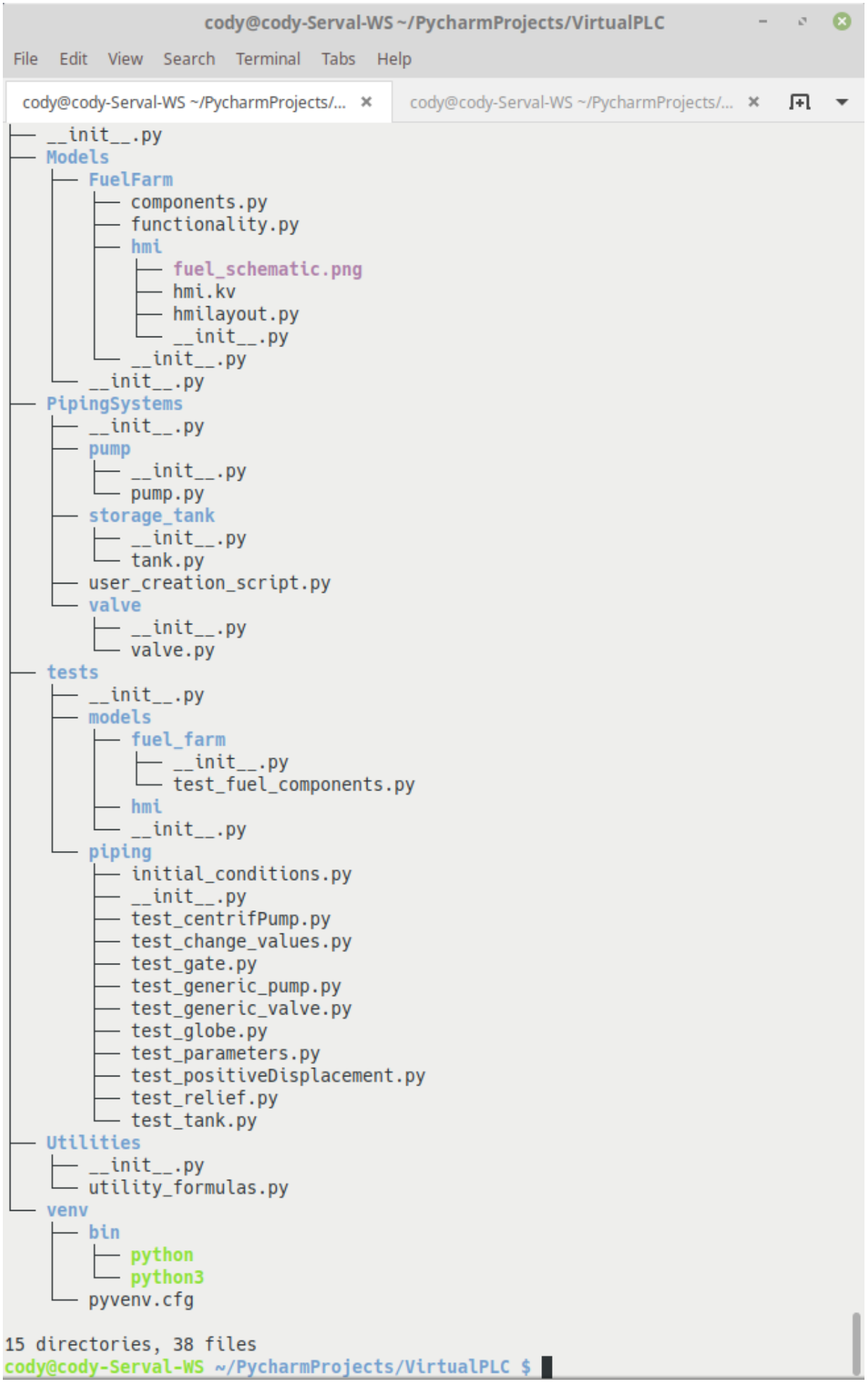
In addition, the fluid characteristics will be based on JP-8/Jet A aviation fuel. It is a kerosene-based fuel that is commonly used by military jets, as well as commercial airliners. While we don't currently model the viscosity of fluids (a key factor if we want to show what happens as temperature fluctuates), we do

account for density and specific gravity; of course, new characteristics can always be added at a later time, if necessary.

Directory structure

One thing that hasn't been mentioned yet is the directory structure for this project. So far, the assumption has been that you are placing all of these items within the same directory, or have an intelligent way to separate files, as in the case of tests.

The following screenshot shows the current directory structure as created on my computer:


cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC
File Edit View Search Terminal Tabs Help
cody@cody-Serval-WS ~/PycharmProjects/...
cody@cody-Serval-WS ~/PycharmProjects/...
├── __init__.py
├── Models
│   ├── FuelFarm
│   │   ├── components.py
│   │   ├── functionality.py
│   │   ├── hmi
│   │   │   ├── fuel_schematic.png
│   │   │   ├── hmi.kv
│   │   │   ├── hmilayout.py
│   │   │   └── __init__.py
│   │   └── __init__.py
│   └── __init__.py
├── PipingSystems
│   ├── __init__.py
│   ├── pump
│   │   ├── __init__.py
│   │   └── pump.py
│   ├── storage_tank
│   │   ├── __init__.py
│   │   └── tank.py
│   ├── user_creation_script.py
│   └── valve
│   ├── __init__.py
│   └── valve.py
├── tests
│   ├── __init__.py
│   ├── models
│   │   ├── fuel_farm
│   │   │   ├── __init__.py
│   │   │   └── test_fuel_components.py
│   │   ├── hmi
│   │   │   └── __init__.py
│   │   └── piping
│   │   ├── initial_conditions.py
│   │   ├── __init__.py
│   │   ├── test_centrifPump.py
│   │   ├── test_change_values.py
│   │   ├── test_gate.py
│   │   ├── test_generic_pump.py
│   │   ├── test_generic_valve.py
│   │   ├── test_globe.py
│   │   ├── test_parameters.py
│   │   ├── test_positiveDisplacement.py
│   │   ├── test_relief.py
│   │   └── test_tank.py
├── Utilities
│   ├── __init__.py
│   └── utility_formulas.py
└── venv
    ├── bin
    │   ├── python
    │   └── python3
    └── pyvenv.cfg
15 directories, 38 files
cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC $

Project directory tree

This output was provided by the Linux `tree` command, and it shows all important directories and files from the project's root directory. You'll note that there are a lot of `__init__.py` files scattered throughout. Each directory that has an `__init__.py` file is treated as if it contains Python packages, regardless of whether it actually does. This prevents directories with common names, such as OS-specific directories, from hiding valid modules that happen to have the same name.

The `models` directory is designed to hold any fluid model. Currently, we only have the fuel farm model that we have been talking about, so it only contains the `FuelFarm` subdirectory. Within that subdirectory are the two files that we will write in this chapter and an `hmi` subdirectory that we will cover in a later chapter, when we learn how to put a graphical interface to this project.

The `PipingSystems` directory contains the subdirectories for the pump, tank, and valve-modeling files. The `user_creation_script.py` file is a program in development, and will not be discussed in this book.

The `tests` directory contains all the tests for the entire program. The `models` subdirectory contains the tests for any models that we create, while `piping` holds the tests for the initial modeling code.

The `Utilities` directory currently holds the `utility_formulas.py` file we created during our initial modeling, but it can also contain future tools for different models.

Finally, `venv` is the directory that holds all the necessary files and configurations for our `pipenv`virtual Python environment. It is created automatically when we create a new `pipenv`.

Component coding

Since we have already coded the foundation parts of the valves, pumps, and tanks, writing the instances of the specific components for our diagram is comparatively easy, as shown in the code listings that we will explain in this section.

We have split the `components.py` file into the following separate parts, with a discussion of each code listing following its respective part. Part 1 is the same introduction that we've seen before. If you look at the date for this listing compared to the previous ones, you'll see that it took about two months to complete the work on the foundation code. Of course, this work was done by one person, who also had other work projects to handle during the same time frame, but it's also a good example of how much time it takes to make full-blown programming projects, especially if it's a personal project:

`components.py` (part 1)

```
1  #!/usr/bin/env python3
2  """
3  FuelFarm.py
4
5  Purpose: Simulate an aviation fuel storage and transfer system.
6
```

```
7  Author: Cody Jackson
8
9  Date: 6/12/18
10 ################################
11 Version 0.2
12 Added path extension for utility formulas
13 Version 0.1
14 Initial build
15 """
```

We've seen the code in part 2 before as well. We have the normal imports, plus the extension to the system path to ensure that we can actually call the parts of the project that we need without errors. We also specify the unique density and specific gravity values of JP-8 fuel:

`components.py` (part 2)

```
1  import sys
2  sys.path.extend(["/home/cody/PycharmProjects/VirtualPLC"])
3  from Utilities import utility_formulas
4
5  from PipingSystems.pump import pump
6  from PipingSystems.valve import valve
7  from PipingSystems.storage_tank import tank
8
9  # Constants
10 DENSITY = 1.629869
11 SPEC_GRAVITY = 0.840
```

In part 3, we define the specifications for the storage tank instances. The comments also show some of the assumptions that are used to provide the parameters:

`components.py` (part 3)

```
1  # Storage tanks
2  # Assumes 36 ft tall tank w/ 1 million gallon capacity = 27778     gallons per foot
3  # Assumes 16 inch diam transfer piping
4  tank1 = tank.Tank("Tank 1", level=36.0, fluid_density=DENSITY,
spec_gravity=SPEC_GRAVITY, outlet_diam=16, outlet_slope=0.25)
5  tank1.static_tank_press = tank1.level
6  tank1.gravity_flow(tank1.pipe_diam, tank1.pipe_slope, tank1.pipe_coeff)
7
8  tank2 = tank.Tank("Tank 2", level=36.0, fluid_density=DENSITY,
spec_gravity=SPEC_GRAVITY, outlet_diam=16, outlet_slope=0.25)
9  tank2.static_tank_press = tank2.level
10 tank2.gravity_flow(tank2.pipe_diam, tank2.pifunctionalitype_slope,
tank2.pipe_coeff)
```

In part 4, we again provide some comments for the basis of the parameters, then start creating instances of the inlet valves to the pumps. Valve flow coefficients are normally found in manufacturers' data manuals, but we can use the diameter of the valve to roughly calculate the coefficient:

`components.py` (part 4)

```
1  # Pump inlet manifold
2  # 16 inch to 4 inch connections
3  gate1 = valve.Gate("Gate valve 1", sys_flow_in=tank1.flow_out,
press_in=tank1.static_tank_press)
4  gate1.calc_coeff(16)
5
```

```
6  gate2 = valve.Gate("Gate valve 2", sys_flow_in=tank2.flow_out,
press_in=tank2.static_tank_press)
7  gate2.calc_coeff(16)
8
9  gate3 = valve.Gate("Gate valve 3")
10 gate3.calc_coeff(16)
11
12 gate4 = valve.Gate("Gate valve 4")
13 gate4.calc_coeff(16)
```

In part 5, we continue defining the inlet valve instances (note that in part 4, only valves 1-4 are 16 inches in diameter, while the valves in part 5 are 4 inches), then create the fuel pump instances. Note that, as screw-type pumps, we have provided a set RPM value, which determines the volume of fluid displaced per revolution. These values would normally be determined by looking at pump curves provided by the manufacturer:

`components.py` (part 5)

```
1  gate5 = valve.Gate("Gate valve 5")
2  gate5.calc_coeff(4)
3
4  gate6 = valve.Gate("Gate valve 6", sys_flow_in=gate3.flow_out + gate4.flow_out,
5  press_in=gate3.press_out + gate4.press_out)
6  gate6.calc_coeff(4)
7
8  gate7 = valve.Gate("Gate valve 7")
9  gate7.calc_coeff(4)
10
11 # Fuel pumps @ 1480 rpm
12 pump1 = pump.PositiveDisplacement("Pump 1", flow_rate_out=0.0,
pump_head_in=utility_formulas.press_to_head(gate5.press_out), displacement=0.24)
13
14 pump2 = pump.PositiveDisplacement("Pump 2", flow_rate_out=0.0,
pump_head_in=utility_formulas.press_to_head(gate6.press_out), displacement=0.24)
```

Part 6 shows the creation of the instances of the remaining fuel pump, the relief valves, and the pressure-regulating throttle valves:

`components.py` (part 6)

```
1  pump3 = pump.PositiveDisplacement("Pump 3", flow_rate_out=0.0,
pump_head_in=utility_formulas.press_to_head(gate7.press_out), displacement=0.24)
2
3  # Pump outlet manifold
4  relief1 = valve.Relief("Relief 1", sys_flow_in=pump1.flow, flow_coeff=0.81)
5  relief2 = valve.Relief("Relief 2", sys_flow_in=pump2.flow, flow_coeff=0.81)
6  relief3 = valve.Relief("Relief 3", sys_flow_in=pump3.flow, flow_coeff=0.81)
7
8  throttle1 = valve.Globe("Flow Control 1", sys_flow_in=pump1.flow,
press_in=pump1.outlet_pressure, flow_coeff=165)
9  throttle2 = valve.Globe("Flow Control 2", sys_flow_in=pump1.flow,
press_in=pump1.outlet_pressure, flow_coeff=165)
10  throttle3 = valve.Globe("Flow Control 3", sys_flow_in=pump1.flow,
press_in=pump1.outlet_pressure, flow_coeff=165)
```

We finish the file with part 7 by creating instances for the fluid-distribution valves that send the fuel back to the tanks or to the flight line.

Also, rather than creating some self-tests at the end of the program, we use the `pass` statement to allow Python to continue. Functionally, there is no difference between using `pass` and not even having the `if __name__ == "__main__"` code block; having it there makes it easier if we want to add self-test code or other code that should be run if `components.py` is run by itself:

`components.py` (part 7)

```
1  gate8 = valve.Gate("Gate valve 8", sys_flow_in=throttle3.flow_out,
press_in=throttle3.press_out)
2  gate8.calc_coeff(4)
3
4  gate9 = valve.Gate("Gate valve 9", sys_flow_in=throttle1.flow_out,
press_in=throttle1.press_out)
5  gate9.calc_coeff(4)
6
7  gate10 = valve.Gate("Gate valve 10", sys_flow_in=throttle3.flow_out,
press_in=throttle3.press_out)
8  gate10.calc_coeff(4)
9
10  if __name__ == "__main__":
11 pass
```

Functionality coding

Now that the instances for all the components are available, we can write the code to actually allow the scenario to work. In this section, the `functionality.py` file is split into separate parts, with a discussion of each code listing following its respective part.

We've seen the code in part 1 many times before, so there is no need to discuss it in detail:

`functionality.py` (part 1)

```
1  #!/usr/bin/env python3
2  """
3  FuelFarm_functionality.py

4
5  Purpose: Ensure valve/pump changes are passed to the rest of the system.
6
7  Author: Cody Jackson
8
9  Date: 6/18/18
10 ################################
11 Version 0.2
12 Added path extension for utility formulas
13 Version 0.1
14 Initial build
15 """
```

In part 2, we import the necessary modules and update the system path. We also list all the parameters and components that are affected by opening or closing valve 1 in lines 7-12. In line 8, we call the `ffc.gate1.open()` method, which is actually from `components.py`.

Next, we check to see which storage tank has a higher level (line 9), and therefore a higher pressure at the outlet of the tank. This is a factor if both tanks are aligned to provide fuel at the same time an unlikely condition, but possible; otherwise, only the tank that is on service will be providing the fuel, so only its gravity flow rate and pressure will be important.

If tank 2 has a higher level/pressure than tank 1, there will be no flow coming from tank 1, and the pressure through valve 3 will be the same as through valve 4. (Hydraulic pressure is the same everywhere within a fluid, which is one reason why liquids are not compressible.) However, a check valve that isn't represented on the schematic is assumed to be in the system after valve 1, so even if tank 1 is empty, there will be no flow through valve 3 (unless valve 5 is open):

`functionality.py` (part 2)

```
1  import sys
2  sys.path.extend(["/home/cody/PycharmProjects/VirtualPLC"])
3  from Utilities import utility_formulas
4  import Models.FuelFarm.components as ffc
5
6  # Gate valve 1
7  def gate1_open():
8  ffc.gate1.open()
9  if ffc.tank2.static_tank_press > ffc.tank1.static_tank_press:
10     ffc.gate1.flow_in = ffc.gate1.flow_out = 0.0
11     ffc.gate3.press_in = ffc.gate4.press_out
12     ffc.gate3.flow_in = 0.0 # No flow because of check valves after valves 1 and 2
```

In part 3, if the level in tank 1 isn't lower than tank 2, then the flow and pressure through valves 3 and 5 will be the same as that which out of valve 1, which is really just the gravitational flow/pressure on the fluid in the tank.

This leads to the **TODO** entry in line 13. Right now, the system checks the tank level, and that determines which tank actually supplies the system. But there is nothing to ensure that the flow comes from a tank with a lower level if that one is deemed "on service," and therefore providing fuel to the rest of the system.

Note that, while pressure is a given, because of hydraulic principles, the flow rate is not guaranteed. Obviously, a valve has to be open for flow to occur, but there will be a certain pressure/flow rate seen at the inlet to each valve. We have to know what these values are to later calculate outlet values, regardless of whether the valve is open or not.

We finish part 3 by writing the method to close gate 1. This not only closes the valve, but also affects the downstream valve pressures and flow, which is described in lines 9-12:

`functionality.py` (part 3)

```
1  else:
2      ffc.gate3.press_in = ffc.gate1.press_out
3      ffc.gate3.flow_in = ffc.gate1.flow_out
4      ffc.gate5.press_in = ffc.gate1.press_out
5      ffc.gate5.flow_in = ffc.gate1.flow_out
```

```
6
7  def gate1_close():
8      ffc.gate1.close()
9      ffc.gate3.press_in = ffc.gate4.press_out
10     ffc.gate3.flow_in = ffc.gate4.flow_out
11     ffc.gate5.press_in = ffc.gate3.press_out
12     ffc.gate5.flow_in = ffc.gate3.flow_out
13     # TODO: ensure that one tank on service allows flow
```

Valve 2 is the same as valve 1, so the code in part 4 shows the same functionality, except the valve numbers have changed:

`functionality.py` (part 4)

```
1  # Gate valve 2
2  def gate2_open():
3      ffc.gate2.open()
4      if ffc.tank2.static_tank_press < ffc.tank1.static_tank_press:
5          ffc.gate2.flow_in = ffc.gate2.flow_out = 0.0
6          ffc.gate4.press_in = ffc.gate3.press_out
7          ffc.gate4.flow_in = 0.0 # No flow because of check valves after valves 1
and 2
8      else:
9          ffc.gate4.press_in = ffc.gate2.press_out
10         ffc.gate4.flow_in = ffc.gate2.flow_out
11         ffc.gate7.press_in = ffc.gate2.press_out
12         ffc.gate7.flow_in = ffc.gate2.flow_out
```

In part 5, we finish with the close method for valve 2. Valves 3 and 4 are opposites, so the discussion about valve 3 applies to valve 4.

With valve 3, we again have to figure out which tank's side has a higher pressure. After we open the valve in line 3, we check to see whether valves 1, 2, and 4 are open at the same time (line 11)—in other words, whether both tanks 1 and 2 are lined up to supply fuel. If so, then we have to figure out which side has the higher pressure (line 12). Based on that information, we can figure out what the inlet pressure to the various valves is:

`functionality.py` (part 5)

```
1  def gate2_close():
2      ffc.gate2.close()
3      ffc.gate4.press_in = ffc.gate4.press_out
4      ffc.gate4.flow_in = ffc.gate4.flow_out
5      ffc.gate7.press_in = ffc.gate4.press_out
6      ffc.gate7.flow_in = ffc.gate4.flow_out
7
8  # Gate valve 3
9  def gate3_open():
10     ffc.gate3.open()
11     if ffc.gate1.position == 100 and ffc.gate2.position == 100 and
ffc.gate4.position == 100: # dual input
12         if ffc.gate3.press_out > ffc.gate4.press_out: # pressure
# from tank 1 > tank 2
13             ffc.gate6.press_in = ffc.gate3.press_out
14             ffc.gate4.press_in = ffc.gate3.press_out
```

Part 6 continues with valve 3. Line 1 checks whether tank 2 has a higher level and pressure than tank 1, while line 4 assumes that the pressure is equal from both tanks, so valves 3 and 4 will show the same

pressure values. In this case, we arbitrarily determine which valve's outlet pressure provides the inlet pressure to valve 6 (line 5).

Line 7 checks that there is no flow coming from the tanks, and then ensures that all parameters for valve 3 are set to zero in line 8. This is just to explicitly ensure that no weird values are available for other calculation input, which could generate hard-to-find errors.

In line 9, if the outlet of tank 2 is closed, then valve 3 will provide input values to valve 4. In line 12, the opposite occurs:

`functionality.py` (part 6)

```
1         elif ffc.gate3.press_out < ffc.gate4.press_out: # pressure           # from
tank 1 < tank 2
2             ffc.gate3.press_in = ffc.gate4.press_out
3             ffc.gate6.press_in = ffc.gate4.press_out
4         else: # Pout from valves 3 and 4 is equal
5             ffc.gate6.press_in = ffc.gate3.press_out # doesn't matter
# which Pout to use
6             ffc.gate6.flow_in = ffc.gate3.flow_out + ffc.gate4.flow_out # combined
flow from valves 3 and 4
7         if ffc.gate1.position == 0 and (ffc.gate2.position == 0 or
ffc.gate4.position == 0): # no input flow
8             ffc.gate3.press_in = ffc.gate3.flow_in = ffc.gate3.press_out =
ffc.gate3.flow_out = 0.0 # Ensure null values
9         if ffc.gate2.position == 0: # valve 3 provides flow to valve 4
10             ffc.gate4.press_in = ffc.gate3.press_out
11             ffc.gate4.flow_in = ffc.gate6.flow_in = ffc.gate3.flow_out
12         if ffc.gate1.position == 0: # valve 4 provides flow to valve 3
13             ffc.gate5.press_in = ffc.gate3.press_out
14             ffc.gate5.flow_in = ffc.gate6.flow_in = ffc.gate3.flow_out
```

In part 7, we determine the parameters if valve 3 is closed. This is comparatively easy, as we only have to account for values from tank 2, as it is the only supply to valves 4 and 6 when valve 3 is closed:

`functionality.py` (part 7)

```
1  def gate3_close():
2      ffc.gate3.close()
3      ffc.gate6.press_in = ffc.gate4.press_out
4      ffc.gate6.flow_in = ffc.gate4.flow_out
5      if ffc.gate2.position == 0:
6          ffc.gate4.press_in = 0.0
7          ffc.gate4.flow_in = 0.0
```

As mentioned before, valve 4 is a mirror image of valve 3, so its code is not presented here; however, the full `functionality.py` file is available in the book's code repository.

Valves 5-7 are pretty similar, the only difference being in the pump they supply. Therefore, only the code for valve 5 is presented in part 8, as shown in the following code:

`functionality.py` (part 8)

```
1  # Gate valve 5
2  def gate5_open():
3      ffc.gate5.open()
4      ffc.pump1.head_in = utility_formulas.press_to_head(ffc.gate5.press_out)
```

```
5
6  def gate5_close():
7      ffc.gate5.close()
8      ffc.pump1.head_in = 0.0
```

Valves 8-10 are similar as well, so part 9 only shows the methods for valve 8.

Valves 8-10 don't have any components that rely on their outlet parameters, so simple calls to their open/close methods are sufficient. However, in a later revision, having some way to account for valves 8 and 9 recirculating flow back to the tanks should be added:

`functionality.py` (part 9)

```
1  # Gate valve 8
2  def gate8_open():
3      ffc.gate8.open()
4
5  def gate8_close():
6      ffc.gate8.close()
```

In part 10, we create a method that will change the tank level and subsequent static pressure. This can be used to programmatically adjust levels as fuel is used, though this particular functionality isn't implemented in this version:

`functionality.py` (part 10)

```
1  # Change tank level
2  def change_tank_level(tank, level):
3      tank.level = level
4      tank.static_tank_press = tank.level
5      if tank == ffc.tank1:
6          ffc.gate1.press_in = ffc.tank1.static_tank_press
7      elif tank == ffc.tank2:
8          ffc.gate2.press_in = ffc.tank2.static_tank_press
9      else:
10  return "Invalid tank number."
```

The fuel pump code is presented in part 11. We already know the speed of the pump, so when the pump is started, we only need to change the speed. The `adjust_speed()` method handles all the parameter changes for us. The only other parameters to account for are primary valves on the outlet.

Pumps 2 and 3 can supply fuel to a number of common valves, notably valves 8 and 10. Normally, the valves wouldn't be supplied by two pumps at once, but if we want to do this, we only have to account for adding the flow rates from pumps 2 and 3. The outlet pressure of all pumps is held constant by their respective regulating valves, which adjust the throttle percentage based on system changes, so a constant pressure value is seen downstream of the pumps.

Because pumps 2 and 3 are functionally the same, only the code for pump 2 is displayed in the following code:

`functionality.py` (part 11)

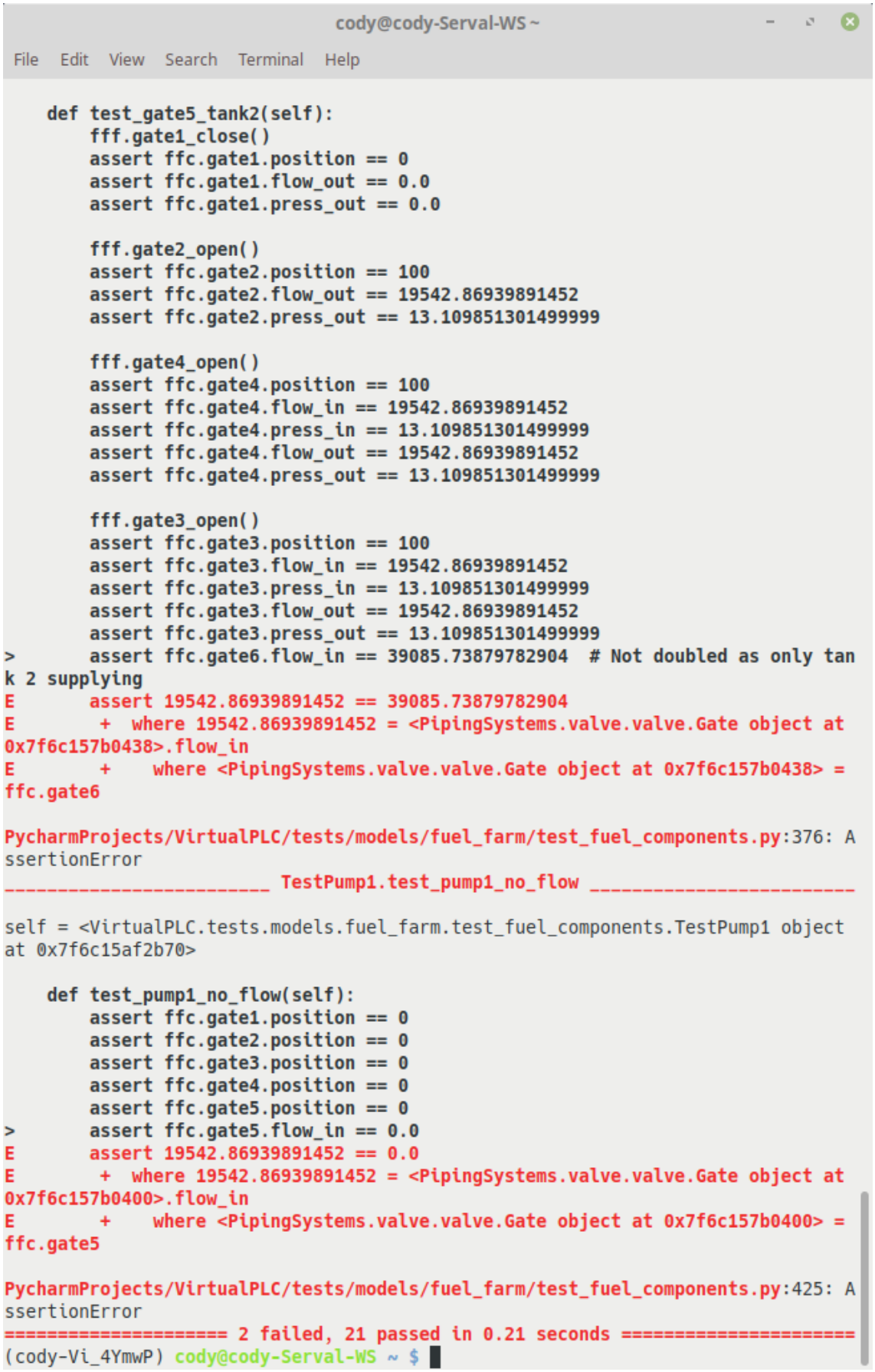
```
1  # Pump 1
2  def pump1_on():
3      ffc.pump1.adjust_speed(1480)
```

```
4       ffc.pump1.outlet_pressure = ffc.gate9.press_in = 50
5       ffc.gate9.flow_in = ffc.pump1.flow
6
7   def pump1_off():
8       ffc.pump1.adjust_speed(0)
9       ffc.pump1.outlet_pressure = 0
10
11   # Pump 2
12   def pump2_on():
13       ffc.pump2.adjust_speed(1480)
14       ffc.pump2.outlet_pressure = ffc.gate8.press_in = ffc.gate10.press_in = 50
15       ffc.gate8.flow_in = ffc.gate10.flow_in = ffc.pump2.flow + ffc.pump3.flow
16
17   def pump2_off():
18       ffc.pump2.adjust_speed(0)
19       ffc.pump2.outlet_pressure = 0
```

Testing

There are a lot of tests that can be performed. The test file for this model, `test_fuel_components.py`, currently holds more than 500 lines of code, so I won't be covering it here; however, it will be included in the book's code repository for you to examine at your leisure.

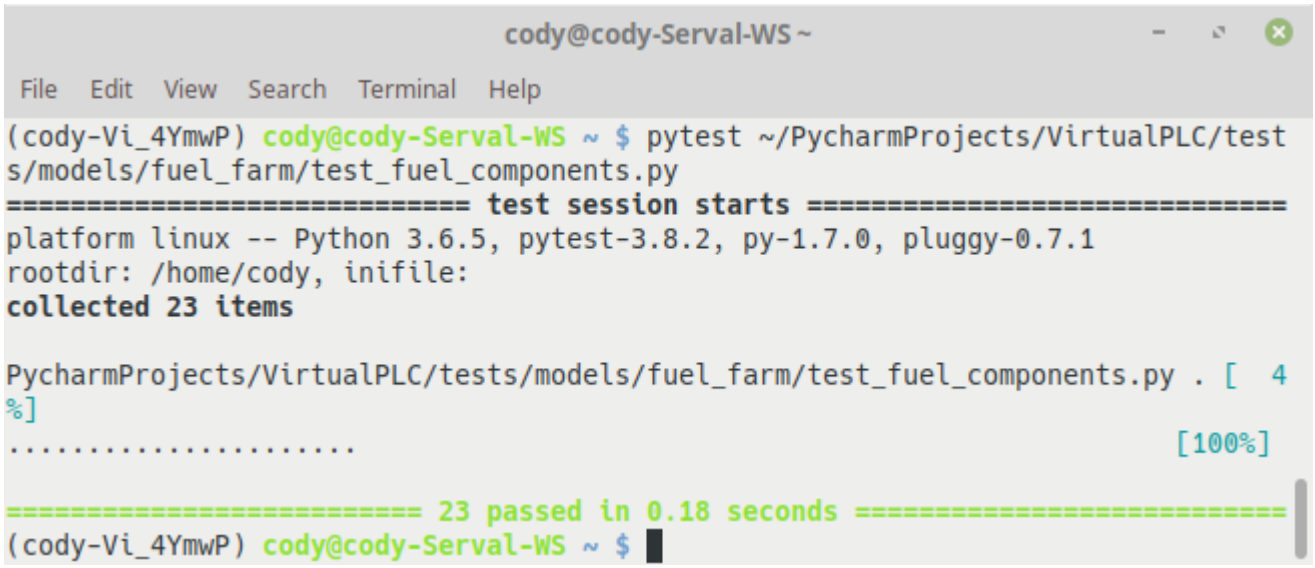
I do want to mention that, quite often (in my experience), testing failures aren't indicative of a problem with the code being tested, but with the tests themselves. For example, look at the following figure:



Fuel testing errors

In this test, two errors are generated. Are they related? Perhaps; a single issue can cause multiple errors, so by fixing and correcting the single issue, multiple errors can be resolved.

In this case, the error being generated in `test_gate5_tank2()` tells us that the expected value (`39085.73879782904`) does not equal the real value (`19542.86939891452`). As noted in the code where the error occurred, the flow rate should be from a single tank, and therefore should not be doubled. If we change the expected value to be the flow rate from a single tank, we receive the test results as shown in following figure:


cody@cody-Serval-WS ~
File Edit View Search Terminal Help
(cody-Vi_4YmwP) cody@cody-Serval-WS ~ $ pytest ~/PycharmProjects/VirtualPLC/tests/models/fuel_farm/test_fuel_components.py
============================= test session starts ==============================
platform linux -- Python 3.6.5, pytest-3.8.2, py-1.7.0, pluggy-0.7.1
rootdir: /home/cody, inifile:
collected 23 items
PycharmProjects/VirtualPLC/tests/models/fuel_farm/test_fuel_components.py . [4%]
....
[100%]
========================== 23 passed in 0.18 seconds ===========================
(cody-Vi_4YmwP) cody@cody-Serval-WS ~ $
Fuel testing success

By making that one change, both errors go away, so we don't have to bother troubleshooting the error in `test_pump1_no_flow()`. This doesn't always happen, as sometimes each error is caused by different issues. However, there are times when a single issue causes multiple, cascading errors.

This demonstrates that you need to have tests that check nearly every possible condition, as well as test the same thing multiple times. Of course, you have to determine which tests are most useful, as well as when to stop; there is such a thing as testing too much, because the more code you have, the more likely you are to introduce bugs. Test only what needs to be tested, and leave the simple stuff alone. You can always refactor it in the next version, if necessary.

Summary

In this chapter, we determined the project requirements to actually make a fuel transfer simulation. With this information, we wrote the code to create the component instances and the system functionality. This allowed us to show the fuel flow from the storage tanks to the pump outlets via different valve lineups. Finally, we talked about testing the simulation and how a single fix to one error can actually fix a variety of issues at once.

In the next chapter, we will look at a software project post-production. We will specifically look at how to document code with docstrings, how to generate user-friendly documentation, and how to review lessons that we will have learned from this project.

Software Post-Production

Once a project is deemed complete (you can always revisit it later with a revision), there are a couple of housekeeping things you should think about. While we added a few comments and basic docstrings to our code, more complete in-code documentation never hurts. By doing that, we can set the project up to help autogenerate user documentation. Finally, it's important to conduct a lessons-learned review of the project to see what went wrong and what could be improved in the future.

In this chapter, we will cover the following topics:

- Writing more complete docstrings
- Generating Sphinx documentation
- Conducting an after-action review

Docstrings

We've mentioned docstrings before, but now we will cover them in greater detail. Docstrings are triple-quoted strings that have special significance within Python. When used, they form the `__doc__` attribute of an object. There are many examples of projects that don't use docstrings, but it is highly advised to incorporate docstrings into your projects. If you do use them, review PEP 257 -- Docstring Conventions (`https://www.python.org/dev/peps/pep-0257/`) to see how to do them right; a **Python Enhancement Proposal** (**PEP**) is used to discuss changes to the Python language. Not following the guidelines is fine, as long as you're consistent within your code. However, if you try to use tools such as Docutils, you can have problems, as they expect the docstrings to be properly formatted; Docutils is a text processing system that converts plain text into formatted documents, such as HTML, and XML.

Docstrings are the very first item in a module, function, class, or method; if they are put elsewhere, chances are good that tools won't recognize them as docstrings. They have to be enclosed by a trio of either single or double-quote marks; that, in conjunction with their location at the beginning of each object, tells Python that they are docstrings and not ordinary triple-quoted strings.

Docstrings can be placed on a single line or, since they are triple-quoted strings, they can also spread across multiple lines. Normally, single lines are used to summarize a basic object while multi-line docstrings can provide more information about an object, such as its expected arguments or output parameters or even what the object is expected to do.

There are a lot of details about docstrings that we won't cover here but are explained in more detail in PEP 257. Some examples include what information to include in a class docstring, especially when creating subclasses, as well as information to include about a function or method. The key takeaway is that docstrings help document and define parts of Python programs and should be included whenever possible.

Related to docstrings are doctests. These are handled by the `doctest` module and function like unit tests, except they are created within a docstring. Doctests are best used as a way to keep docstrings up-to-date and ensure the code actually does what the documentation says it should. Doctests utilize the interactive Python interpreter to perform the tests, rather than run as separate test files. The following is an example of a doctest included within the docstring of a function:

```
def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """
```

We won't cover all of the docstrings for all of the code we have written, but the complete files in this book's code repository does include them. However, we will show the docstrings for `valve.py` as a representative example of how docstrings could be written. Note that we won't reprint the entire file or the actual code logic, but just the key objects that benefit from expanded docstrings.

In part 1 of the following code, we create the docstring for the parent `Valve` class. The first line is a basic summary of the class; this should be written as a single-line summary statement, in case we decide not to include any additional information.

Line 4 provides additional information about the class, specifically what the `Cv` parameter is and what it represents for a valve. Line 6 lists all of the parameters identified in the class, while line 8 lists the methods that are part of the class. Notice that we don't list getter/setter methods for valve positions, as those have been converted in to class properties:

```
# valve.py docstrings (part 1)
1class Valve:
2    """Generic class for valves.
3
4    Cv is the valve flow coefficient: number of gallons per minute at 60F through a
fully open valve with a press. drop of 1 psi. For valves 1 inch or less in diameter,
Cv is typically < 5.
5
6    Variables: name, position, Cv, deltaP, flow_in, flow_out, press_out, press_in
7
```

```
8    Methods: calc_coeff(), press_drop(), valve_flow_out(), get_press_out(), open(),
close()
9    """
```

When we create the initialization method in part 2 in the following code, it is useful to provide the parameters that are being initialized (lines 4-10). These parameter statements include the name of the parameters, as well as a short sentence about what the parameter represents:

```
# valve.py docstrings (part 2)
1    def __init__(self, name="", sys_flow_in=0.0, sys_flow_out=0.0, drop=0.0,
position=0, flow_coeff=0.0, press_in=0.0):
2        """Initialize valve.
3
4        :param sys_flow_out: Fluid flow out of the valve
5        :param drop: Pressure drop across the valve
6        :param press_in: Pressure at valve inlet
7        :param name: Instance name
8        :param sys_flow_in: Flow rate into the valve
9        :param position: Percentage valve is open
10       :param flow_coeff: Affect valve has on flow rate; assumes a 2 inch, wide open
valve
11        """
```

In part 3 of the following code, we provide not only an argument parameter for the `calc_coeff()` method, but also indicate what the return object represents. You can provide the name/type of the return value, but it isn't necessary, as the code itself should be indicative enough:

```
# valve.py docstrings (part 3)
1    def calc_coeff(self, diameter):
2        """Roughly calculate Cv based on valve diameter.
3
4        :param diameter: Valve diameter
5
6        :return: Update valve flow coefficient
7        """
8
9    def press_drop(self, flow_out, spec_grav=1.0):
10        """Calculate the pressure drop across a valve, given a flow rate.
11
12        Pressure drop = ((system flow rate / valve coefficient) ** 2) * spec.
gravity of fluid
13
14        Cv of valve and flow rate of system must be known.
```

In part 4, we continue providing assumptions in line 1, then we list the input parameters in lines 3 and 4. With line 6, we provide the exception that is generated within this method, and finish up with lines 8 and 9 that provide the return object information. Note that, in line 9, the return type is provided; again, it's not required, but it might be useful if an actual object is returned, rather than just updated.

We start a new method in line 12, with lines 13 and 15 providing additional information about the method, as is normal for a docstring:

```
# valve.py docstrings (part 4)
1        Specific gravity of water is 1.
2
3        :param flow_out: System flow rate into the valve
4        :param spec_grav: Fluid specific gravity; default assumes fluid is water
5
6        :except ZeroDivisionError: Valve coefficient not provided
```

```
7
8         :return: Update pressure drop across valve
9         :rtype: float
10         """
11
12     def valve_flow_out(self, flow_coeff, press_drop, spec_grav=1.0):
13         """Calculate the system flow rate through a valve, given a pressure drop.
14
15         Flow rate = valve coefficient / sqrt(spec. grav. / press. drop)
```

Part 5 continues the `valve_flow_out()` method by adding the incoming parameter arguments, the method exception, return object, and return type. Line 11 starts a new method:

```
# valve.py docstring (part 5)

1         :param flow_coeff: Valve flow coefficient
2         :param press_drop: Pressure drop (psi)
3         :param spec_grav: Fluid specific gravity
4
5         :except ValueError: Valve coefficient or deltaP <= 0
6
7         :return: Update system flow rate
8         :rtype: float
9         """
10
11     def get_press_out(self, press_in):
12         """Get the valve outlet pressure, calculated from inlet pressure.
13
14         :param press_in:  Pressure at valve inlet
```

In part 6, lines 1-3 complete the `get_press_out()` method docstring. With line 5, we gave the `Gate`valve subclass. Docstrings for subclasses are slightly different, as they identify what the parent class is (line 8) and the methods that are part of the subclass (lines 10-12).

We finish this code listing by starting the `read_position()` method and its initial docstring summary:

```
# valve.py docstring (part 6)
1         :return: Pressure at valve outlet
2         :rtype: float
3         """
4
5class Gate(Valve):
6     """Open/closed valve.
7
8     Subclasses Valve.
9
10     Methods:
11         read_position()
12         turn_handle()
13     """
14     def read_position(self):
15         """Identify the position of the valve.
```

In part 7, we finish the rest of the `read_position()` docstring and then create the docstring data for `turn_handle()`:

```
# valve.py docstring (part 7)
1         :return: Indication of whether the valve is open or closed.
2         :rtype: str
3         """
```

```
4
5    def turn_handle(self, new_position):
6        """Change the status of the valve.
7
8        :param new_position: New valve position
9
10        :return: Update valve position
11        """
```

As these classes and methods are representative of how docstrings can be made, the rest of the file is not provided. However, the complete `valve.py` file and the rest of the code for this project are provided in this book's file repository for review by the reader.

Sphinx documentation

Sphinx was written for the Python documentation and is used extensively in official document creation, though it can be used to create other documents as well. All of the documentation on the Python site is generated by Sphinx and most Python projects use it for their websites. Even the Sphinx website is written in **reStructuredText** (**reST**) and converted in to HTML.

PEP 287 proposes that reST markup should be used for structured text documentation within Python docstrings, PEPs, and other documents that require structured markup. Of course, plain text docstrings are not deprecated; reST simply provides more options for developers who want to be more expressive in their documentation.

Sphinx can convert reST into HTML, PDF, ePub, Texinfo, and man pages. The program is also extensible, providing plug-ins for generating mathematical notation from formulas or highlighting source code.

Sphinx can be installed in the normal ways: either through `pip` or downloading an installation package. Once installed, it is suggested to move (through the command prompt) to the project directory, as the program defaults to looking for files in the current directory. It's not necessary, as the path can be changed later; it just makes life easier.

Run the `sphinx-quickstart` command. The program will run through an interactive, text-based setup, which will ask the user multiple questions. The questions are generally self-explanatory, but be sure to check the Sphinx documentation if something doesn't make sense. Don't panic, however, if you just pick the defaults and you don't get the results expected. This process is simply creating the default configuration files, which can be manually modified later. A key thing to point out is, if you want to use your docstrings to generate your documenation, ensure you select `autodoc` for installation.

When completed, you should now see `conf.py` and `index.rst` in your project directory. These are used to allow Sphinx to operate. `conf.py` as the configuration file for Sphinx. It is the primary location for setting up Sphinx and the entries made during the quickstart process are stored here. `index.rst` is the primary file for telling Sphinx how to create the final documentation. It basically tells Sphinx what modules, classes, and so on to include in the documentation.

By default, `conf.py` looks for files in `PYTHONPATH`; if you are looking to use files in another location, make sure you set it up correctly at the top of the file. Specifically, uncomment `import os`, `import sys` and the `sys.path.insert()` line (and update the path as needed).

This is demonstrated in the following screenshot:

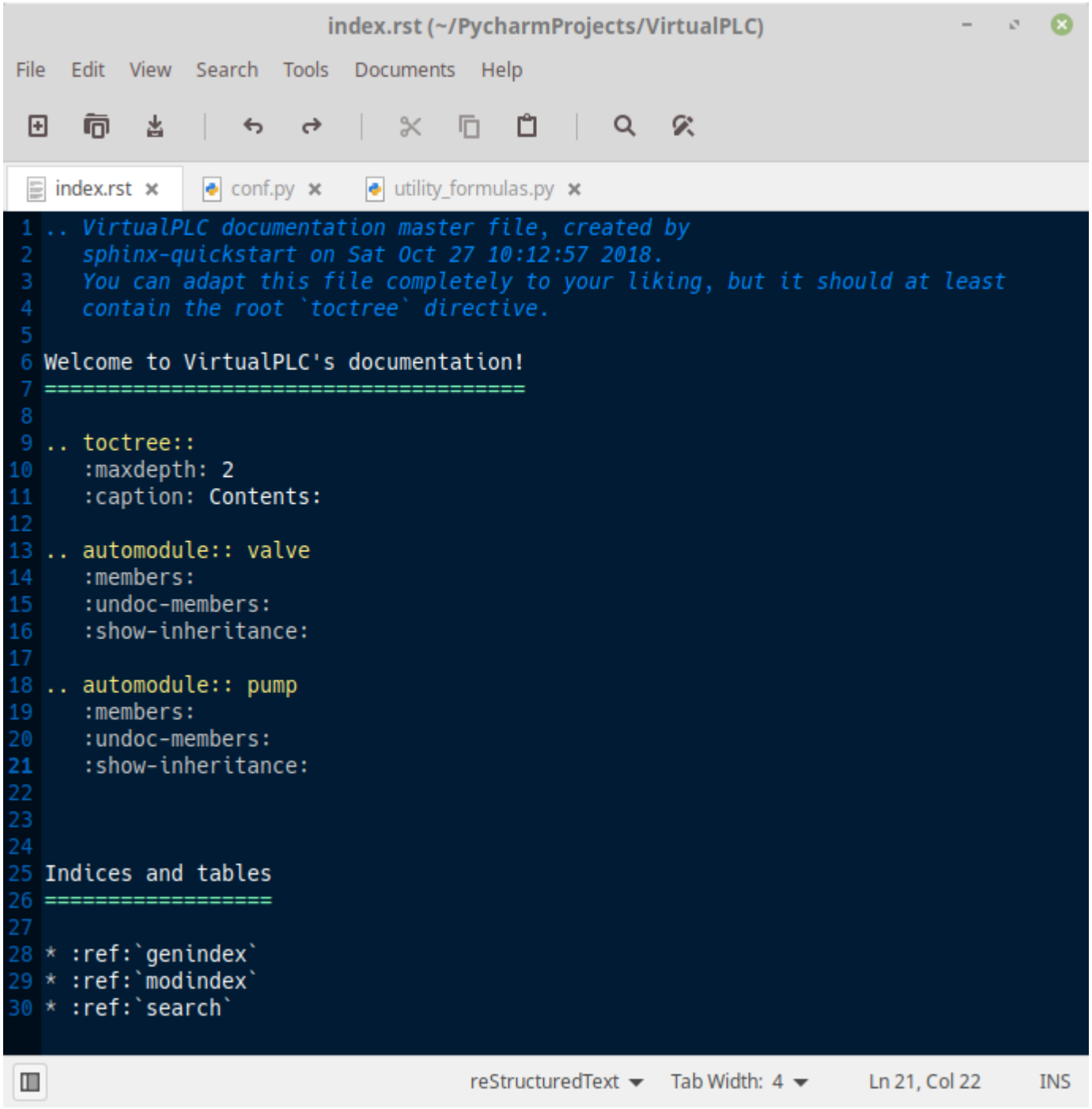
conf.py (~/PycharmProjects/VirtualPLC)
File Edit View Search Tools Documents Help
index.rst
conf.py
utility_formulas.py
-*- coding: utf-8 -*-
#
Configuration file for the Sphinx documentation builder.
#
This file does only contain a selection of the most common options. For a
full list see the documentation:
http://www.sphinx-doc.org/en/stable/config
-- Path setup --
If extensions (or modules to document with autodoc) are in another directory,
add these directories to sys.path here. If the directory is relative to the
documentation root, use os.path.abspath to make it absolute, like shown here.
#
import os
import sys
sys.path.insert(0, os.path.abspath('./PipingSystems/valve'))
sys.path.insert(0, os.path.abspath('./PipingSystems/pump'))
# -- Project information ---
project = u'VirtualPLC'
copyright = u'2018, Cody Jackson'
author = u'Cody Jackson'
# The short X.Y version
version = u''
# The full version, including alpha/beta/rc tags
release = u'0.2'
# -- General configuration ---
# If your documentation needs a minimal Sphinx version, state it here.
#
# needs_sphinx = '1.0'
# Add any Sphinx extension module names here, as strings. They can be
# extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
# ones.
extensions = [
'sphinx.ext.autodoc',
]
Python
Tab Width: 4
Ln 7, Col 45
INS

conf.py

In the preceding screenshot, we have an example `conf.py` file for our fuel farm project. Only two of the component paths have been added for demonstrative purposes. In reality, you would want to add all of the paths that you want to include within the documentation.

If you set up `conf.py` to use `autodoc` (line 43 in the preceding screenshot), the next step is simple. Go to `index.rst` and tell Sphinx to automatically find the information for the documentation. The easiest way to do this is take a look at `http://www.sphinx-doc.org/en/stable/ext/autodoc.html#module-sphinx.ext.autodoc`, which explains how to automatically import all desired modules and retrieve the docstrings from them.

An example of `automodule` from `autodoc` is shown in following screenshot. If you go to the `autodoc` site, you'll see you can also use `autoclass` and `autoexception` to document individual components within a Python program, if you don't want or need to document an entire module. There are also a large number of options and advanced features available, which are beyond the scope of this book. The following screenshot shows the `index.rst` file:



index.rst

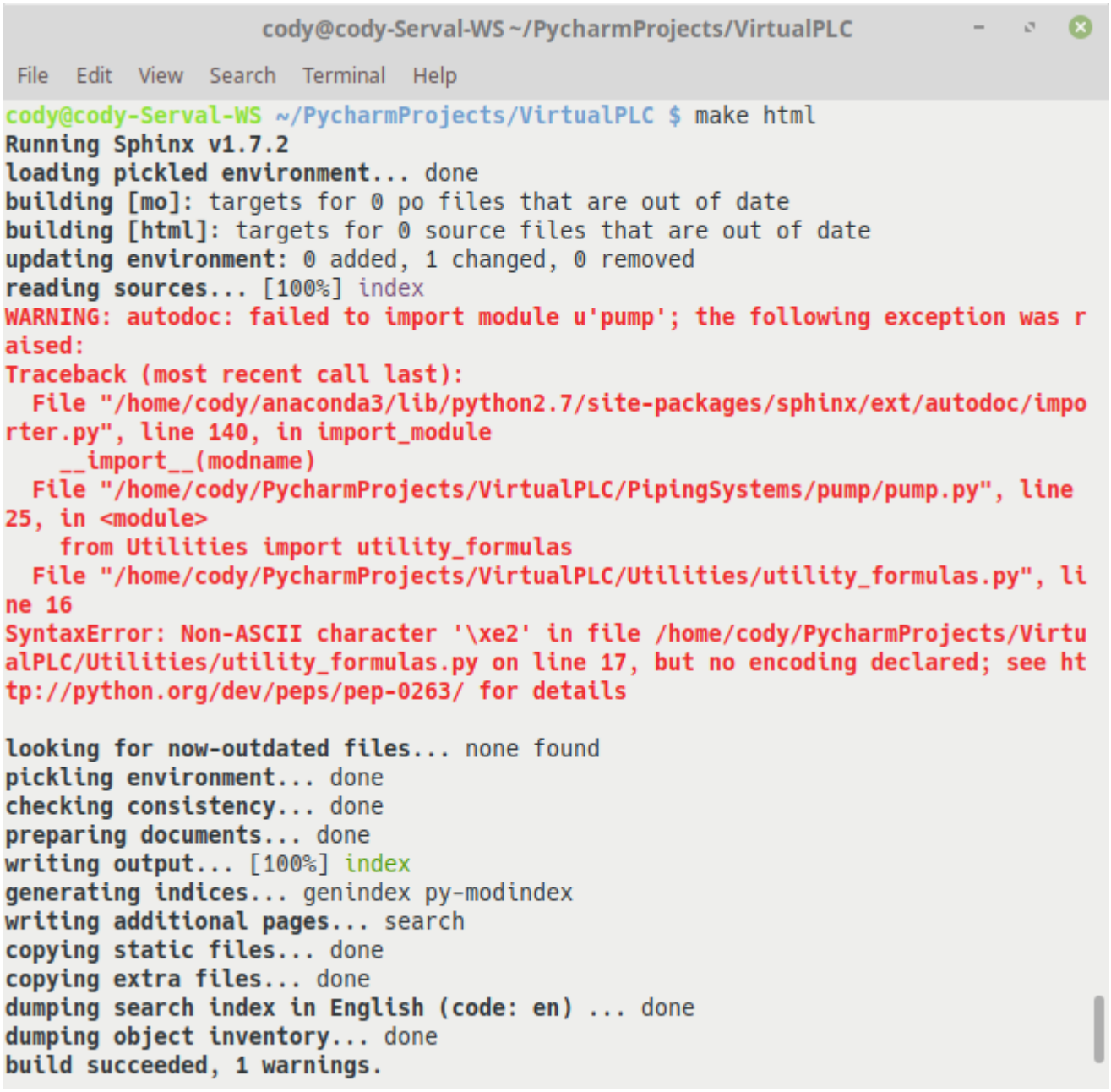
The preceding screenshot shows an example for this project. Everything apart from the two `automodule` sections is default information, automatically created by Sphinx. The `automodule`sections tell Sphinx what the name is of the Python module to import: the Python filename without the `.py` extension. The other parts of the file are described as follows:

- `members`: This automatically gathers documentation for all public classes, methods, and functions that have docstrings. If you don't use it, only the docstring for the main object (a module, in this case) will be imported.
- `undoc-members`: This does the same thing, except it will get objects that don't have docstrings. Obviously, the information for these items will be limited compared to a docstring.

- `show-inheritance`: This specifies that the inheritance tree for the module will be included. Needless to say, if you aren't using inheritance, this won't do much good.

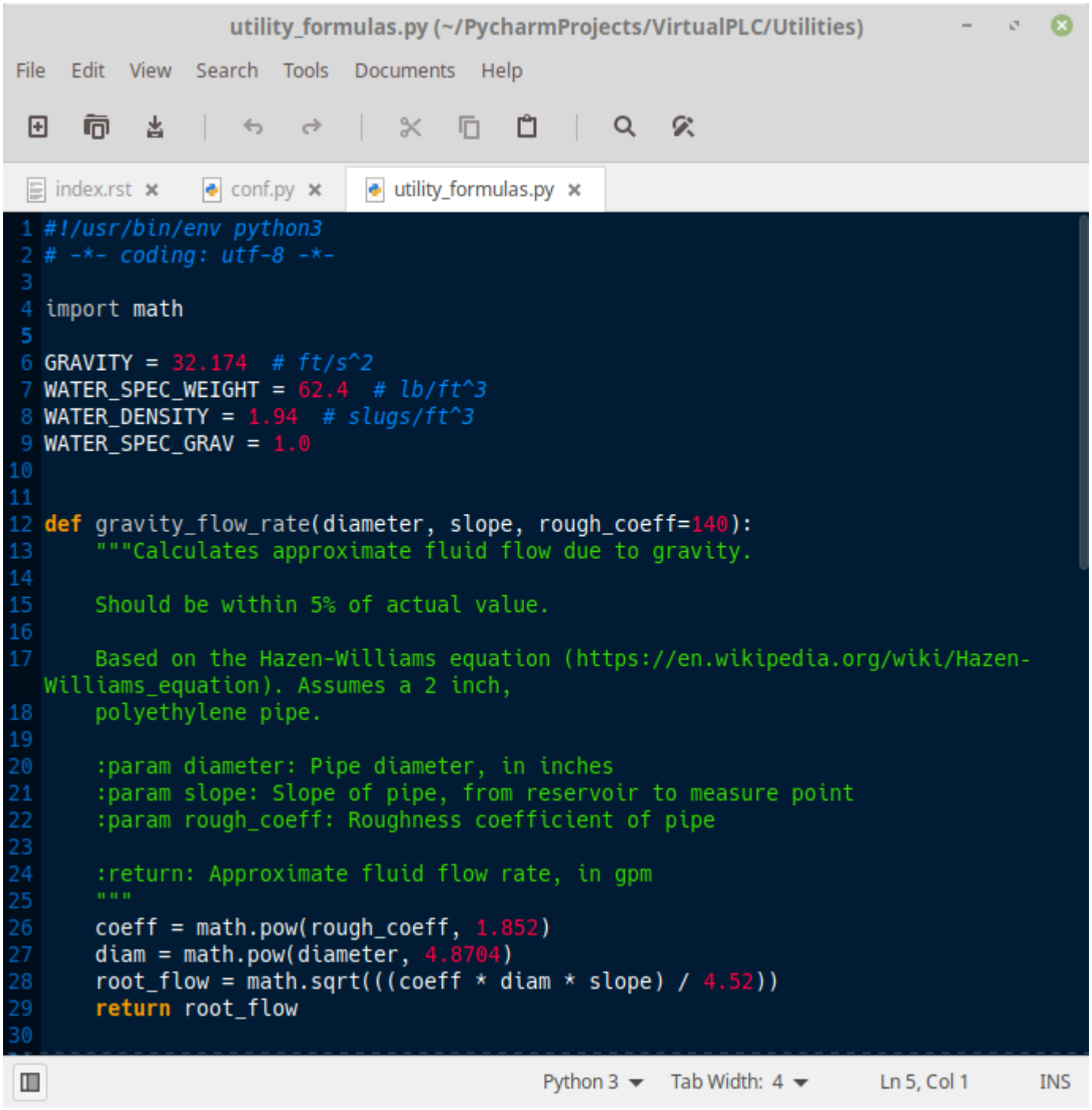
Once `conf.py` and `index.rst` are updated, run the `make html` command to generate the HTML files for the project. It is not unusual to receive errors, particularly Sphinx telling you it can't find a particular module. This usually means you provided the wrong path in `conf.py`.

One unusual error you can receive is shown in the following screenshot:

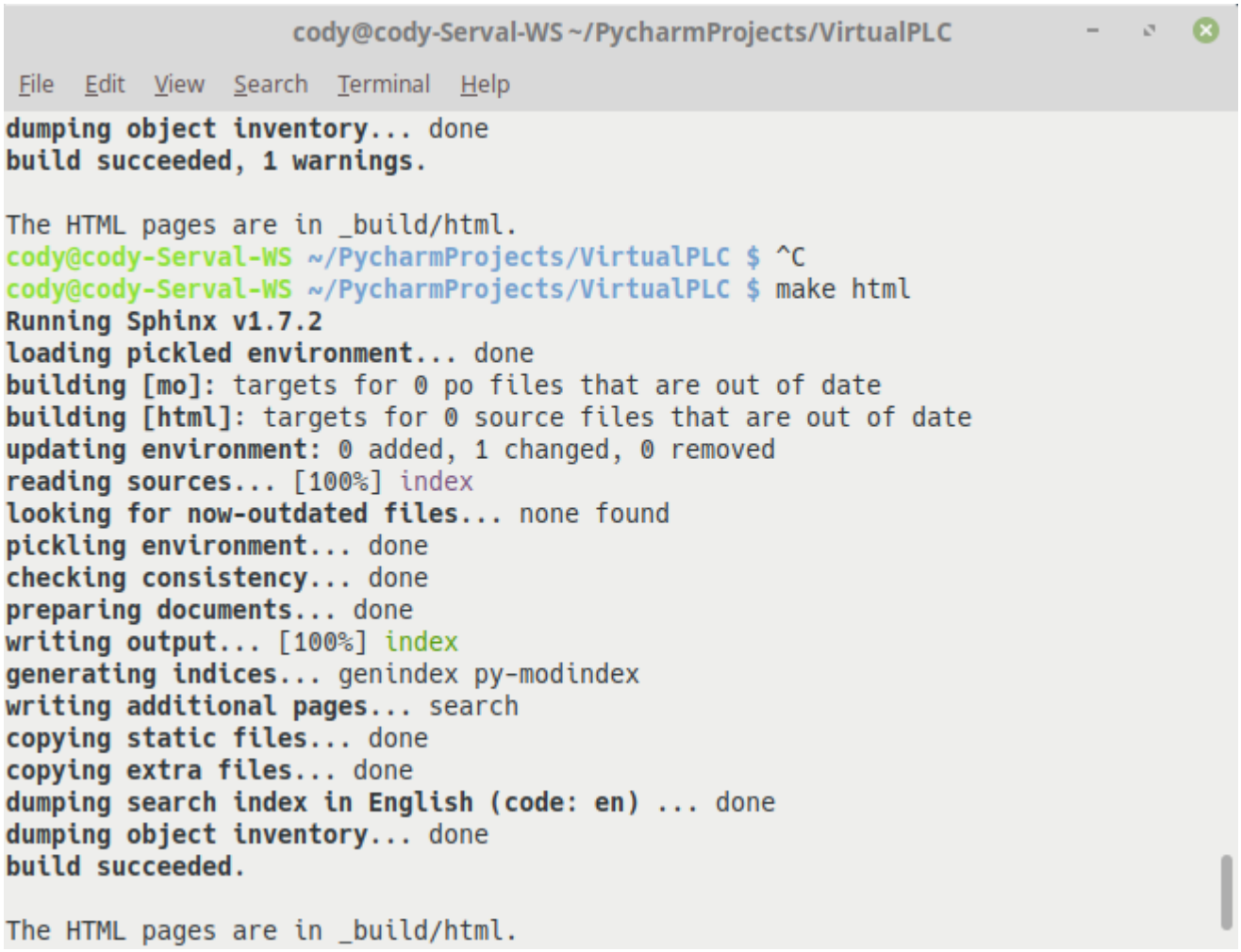

cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC
File Edit View Search Terminal Help
cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC $ make html
Running Sphinx v1.7.2
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 1 changed, 0 removed
reading sources... [100%] index
WARNING: autodoc: failed to import module u'pump'; the following exception was raised:
Traceback (most recent call last):
File "/home/cody/anaconda3/lib/python2.7/site-packages/sphinx/ext/autodoc/importer.py", line 140, in import_module
__import__(modname)
File "/home/cody/PycharmProjects/VirtualPLC/PipingSystems/pump/pump.py", line 25, in <module>
from Utilities import utility_formulas
File "/home/cody/PycharmProjects/VirtualPLC/Utilities/utility_formulas.py", line 16
SyntaxError: Non-ASCII character '\xe2' in file /home/cody/PycharmProjects/VirtualPLC/Utilities/utility_formulas.py on line 17, but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
generating indices... genindex py-modindex
writing additional pages... search
copying static files... done
copying extra files... done
dumping search index in English (code: en) ... done
dumping object inventory... done
build succeeded, 1 warnings.
Sphinx error

This error indicates that Sphinx is unable to proceed because it found an invalid character. If you look at the `utility_formulas.py` file, there is nothing on line 17 that would cause the error. However, if you check line 15, you'll see that we included a percentage symbol.

Rather than removing the symbol, if we modify `utility_formulas.py` to include line 2 in the following screenshot, that will remove the Sphinx error:


utility_formulas.py (~/PycharmProjects/VirtualPLC/Utilities)
File Edit View Search Tools Documents Help
index.rst
conf.py
utility_formulas.py
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 import math
5
6 GRAVITY = 32.174 # ft/s^2
7 WATER_SPEC_WEIGHT = 62.4 # lb/ft^3
8 WATER_DENSITY = 1.94 # slugs/ft^3
9 WATER_SPEC_GRAV = 1.0
10
11
12 def gravity_flow_rate(diameter, slope, rough_coeff=140):
13 """Calculates approximate fluid flow due to gravity.
14
15 Should be within 5% of actual value.
16
17 Based on the Hazen-Williams equation (https://en.wikipedia.org/wiki/Hazen-Williams_equation). Assumes a 2 inch,
18 polyethylene pipe.
19
20 :param diameter: Pipe diameter, in inches
21 :param slope: Slope of pipe, from reservoir to measure point
22 :param rough_coeff: Roughness coefficient of pipe
23
24 :return: Approximate fluid flow rate, in gpm
25 """
26 coeff = math.pow(rough_coeff, 1.852)
27 diam = math.pow(diameter, 4.8704)
28 root_flow = math.sqrt(((coeff * diam * slope) / 4.52))
29 return root_flow
30
Python 3
Tab Width: 4
Ln 5, Col 1
INS
Update utility_formulas.py

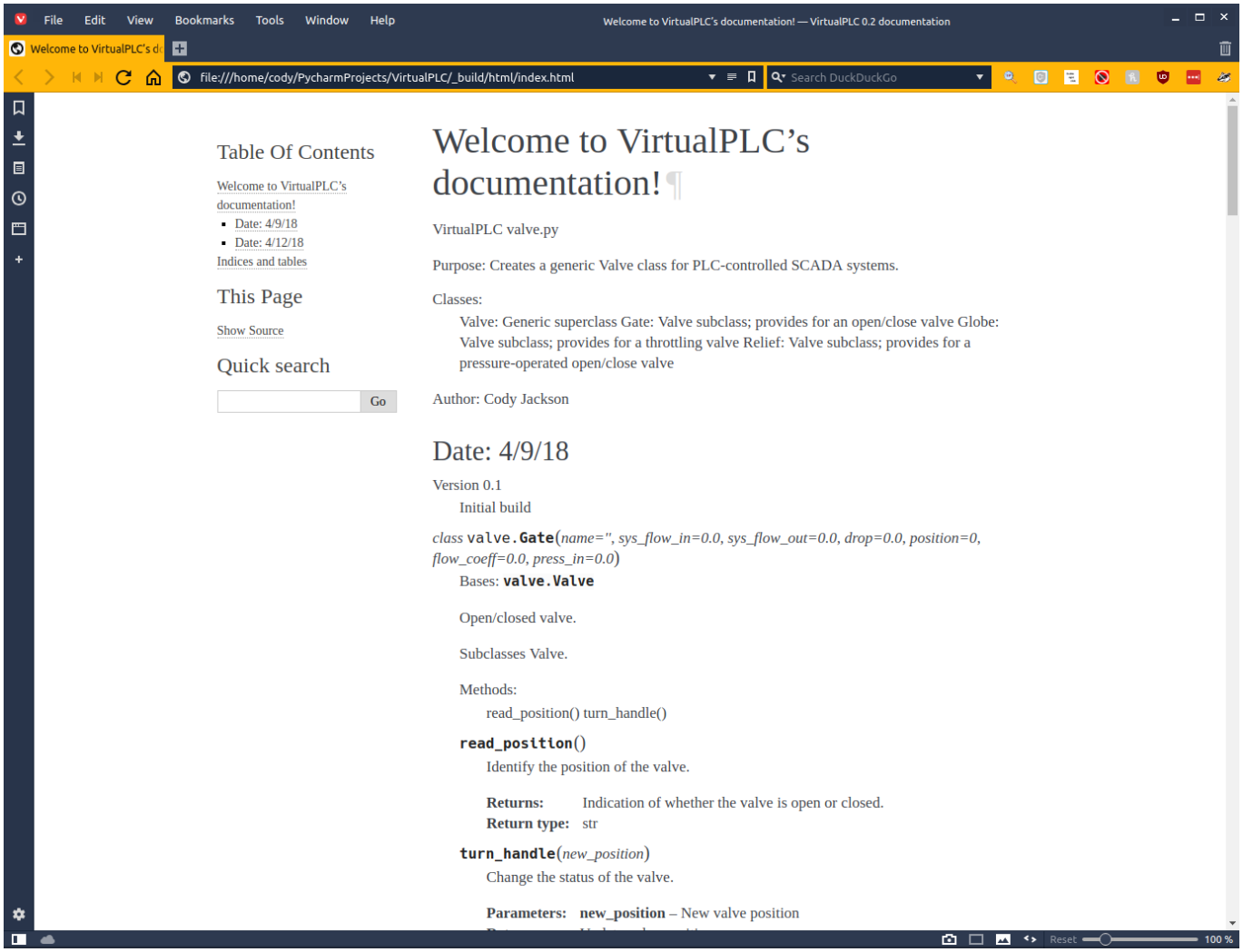
When Sphinx runs correctly, you should see something like the following screenshot:


cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC
File Edit View Search Terminal Help
dumping object inventory... done
build succeeded, 1 warnings.
The HTML pages are in _build/html.
cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC $ ^C
cody@cody-Serval-WS ~/PycharmProjects/VirtualPLC $ make html
Running Sphinx v1.7.2
loading pickled environment... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 1 changed, 0 removed
reading sources... [100%] index
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] index
generating indices... genindex py-modindex
writing additional pages... search
copying static files... done
copying extra files... done
dumping search index in English (code: en) ... done
dumping object inventory... done
build succeeded.
The HTML pages are in _build/html.
Sphinx success

If you go into the project directory, you should find `index.html` in the `_build/html` directory (assuming you used the default values).

When you open it, you should see something like following screenshot. Currently, this output only shows the information in the docstrings, but reST allows you to create much more useful information, such as explaining how to use the software.

If you don't like the default theme, there are a number of other themes included with Sphinx. Obviously, being HTML, you can make your own as well. The following screenshot shows the Sphinx output:



Sphinx output

Lessons learned

Once the project is deemed complete, at least for this particular version, it is a good time to take a step back from programming and planning to review the project and see what went right and what went wrong.

For example, these are some of the things I have learned when writing the fuel farm scenario, in no particular order:

- Don't try to write the parameters into docstrings until the project is done. Refactoring can cause items to change, such as eliminating getter/setter methods and replacing them with properties.
- Write basic docstring summaries while coding, as it helps to clarify what a function/method/class is supposed to do. This helps to ensure that a function or method does only one thing, rather than trying to have multiple operations occurring.

- Update version numbers each day. Even if not using a CI environment, it is helpful to provide a version number to the final code pushed to the repository at the end of the day. If you looked closely at the version information in several of the code files, you'll see that the current version is listed as "0.2". This is because I manually updated the version number arbitrarily based on personal preference.
- While this is perfectly acceptable for personal projects, it isn't really conducive to identify bugs in particular versions, nor does it help when rolling back to previous versions. It also means that the programmer is slightly behind the curve when working in a team, where code is automatically built each night and receives an internal version number.
- Keep `TODO` entries in your code as you think of things. Where they are placed is up to the individual coder, unless there is a company policy. They could all be placed at the beginning or end of a file or they could be placed in the location they apply to.
- Don't think that unit tests that fail are due to bad code. When writing this program, it was discovered that the majority of failed tests were caused by tests receiving incorrect data; that is, the core code was correct, but the tests was requesting the wrong values.
- Keep unit tests up-to-date. This can be very difficult if the code is frequently refactored. This is where doctests can help; since they are included in the docstring, when the docstring is updated, the doctests are part of the update. Alternatively, test-driven development means that the code is written to make the tests pass; once a test passes, no more coding needs to be done, apart from bug fixes.
- Don't be afraid to brute-force solutions. In my philosophy, coding is like flying a plane: any crash you walk away from is a successful landing. In the same way, any code that gets the job done is a success. You can always go back and refactor the code to make it more concise, more efficient, and so on, but you have to have a solution first. Coding effectively comes with practice, and it may not be immediately obvious what the best solution is, so writing your first iteration like a first-year student is perfectly fine.
- Get out of your comfort zone. The fuel farm scenario discussed in this book includes a lot of information that the average person wouldn't know, whether it is the difference between valves or even the mathematical formulas used in the scenario. Most business applications are pretty standard and often come down to capturing data entry and interacting with databases; engineering principles aren't required. However, the more knowledge you have, the more valuable you become to employers. This obviously opens up many more job opportunities—for example, with the knowledge from the fuel farm scenario, you are better positioned to apply for work in the industrial sector, as you can demonstrate that you have a basic understanding of systems engineering; in other words, you can think about how different systems affect other systems, including cascading consequences and real-world applicability. Alternatively, if you want to get into video game design, you should have an idea of some of the concerns developers have when it comes to simulating an environment. Many companies talk about their physics models, such as gravity or ragdoll mechanics. The formulas used in this book are simple algebraic calculations, while video games often use algebraic geometry, matrix calculations, and so on. However, the underlying idea is the same: use real-world physics to determine an outcome.
- Consider alternative implementations of the same thing. For example, while this project used class instances to hold all of the parameter data for each component, an alternative solution could be to use a database. A database could make it easier for other programs to access the system information, such as incorporating the virtual fuel farm into an industrial control simulation that uses Modbus or other industrial protocols to change plant parameters. By using SQLAlchemy, the need to know SQL is reduced, so the code could remain Python-centric. This also means it would be easier to wrap the entire project into a website.

That's about it for lessons learned by me. There are always things to consider, especially when working on a team, as group dynamics, resource allocation, and management support come into play as well.

Summary

In this chapter, we learned how to add useful information to docstrings to enhance their use for future users, how to use Sphinx to autogenerate documentation, and some of the lessons learned from this project.

In the next chapter, we will look at how to take a purely text-based program, like we just created, and make it into a graphical program. We will plan out the graphical interface, look at different frameworks, and make plans for our fuel farm project.

Graphical User Interface Planning

The **graphical user interface** (**GUI**) is what most people now see when they use computers. The macOS and Windows operating systems are primarily graphical-based, as graphical interfaces are easier to use; while you can do some things from a text console, such as Windows PowerShell, the majority of features are accessed through a graphical window.

Currently, our fuel farm project is strictly text-based. To make it easier for people to use, we can put a graphical interface on it, which allows people to click on the components to manipulate them. In the industrial controls world, this is called a human-machine interface, more commonly known as an HMI. An HMI allows an operator to work with a system and change settings without having to manually walk to each component. It also displays system parameters, such as flow rates or pressure values.

In this chapter, we will cover the following topics:

- GUI functionality
- User environment
- GUI frameworks

GUI functionality

Graphical interfaces have to provide usability to users. Many of us have dealt with GUIs that were not well designed; the issue is with either presenting the information in a non-intuitive manner or not providing the tools we need to accomplish a task. Sometimes, the designer of a program never actually uses the program. Often, what makes sense on paper doesn't actually carry over to the final user's interaction.

Some organizations have guidelines for GUI best practices, such as Apple's Human Interface Guidelines (`https://developer.apple.com/design/human-interface-guidelines/macos/overview/themes/`). In Apple's case, they provide directions for designing software for macOS, iOS, the Apple watchOS, and even Apple's tvOS.

Another example is the GNOME desktop environment (`https://developer.gnome.org/hig/stable/`). Other organizations or products have their own recommendations; Wikipedia provides a list of various guidelines on its Human Interface Guidelines page for different environments, as different vendors, operating systems, and software recommend, or require, particular interface configurations.

The purpose of these guidelines is to encourage the developers to use intuitive and consistent interfaces, particularly when designing a particular desktop or device environment. Thus, users can expect to find the same functionality in the same locations across applications.

One thing to recognize is that a GUI is normally a graphical wrapper around text-based commands. Every text command that a user can perform on a command line could also be handled by a GUI. Sometimes, however, it can take multiple user actions, such as mouse clicks, to perform the same thing through a

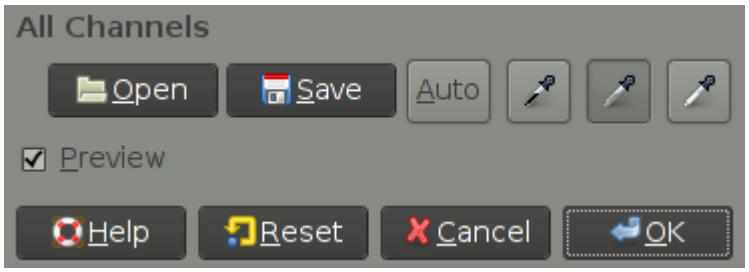
single or a combined line command. Since people are visually oriented, more information can be conveyed via a graphical interface than a person could understand through text.

Most GUI environments use the **Windows**, **Icons**, **Menus**, **Pointer** (**WIMP**) paradigm; this is especially common in traditional desktop/laptop applications, as well as websites. Other GUIs can use different paradigms, such as heads-up displays in video games, or icon-based menus in mobile phones that don't have windows.

GUI elements

When designing and using a GUI, there are a number of common elements (often called widgets) that need to be considered. The following is a non-exhaustive list of widgets, and the options available, which may be limited by the graphical framework used:

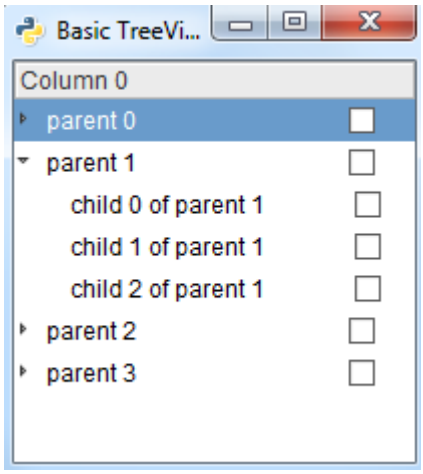
- **Input controls**:
  - **Buttons**: A widget that provides a simple way to trigger an event, such as confirming an action. The following screenshot shows an example of it:



  -

  - **Text fields**: Boxes that accept text input from the user:

- **Checkboxes**: Allows the user to make a binary (on/off) selection for a particular option:

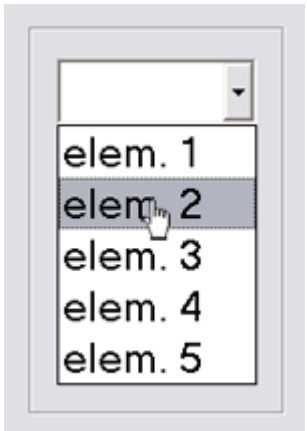


-

- **Radio buttons**: Allows the user to make a single option from a set of mutually exclusive choices:

-

- **Drop-down lists**: Normally shows a single item; when the drop-down arrow is clicked, more items are displayed, though only one item can be selected. The following screenshot shows an example of it:

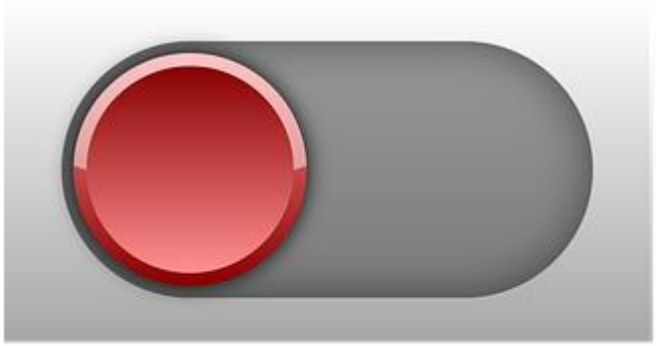


-

- **List boxes**: Similar to drop-down lists, list boxes allow one or more choices to be made. The following screenshot shows an example of it:

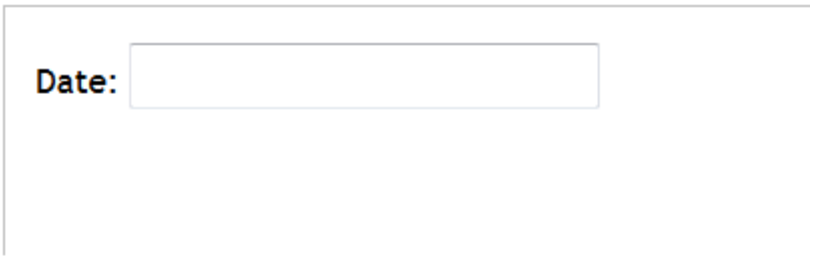
-

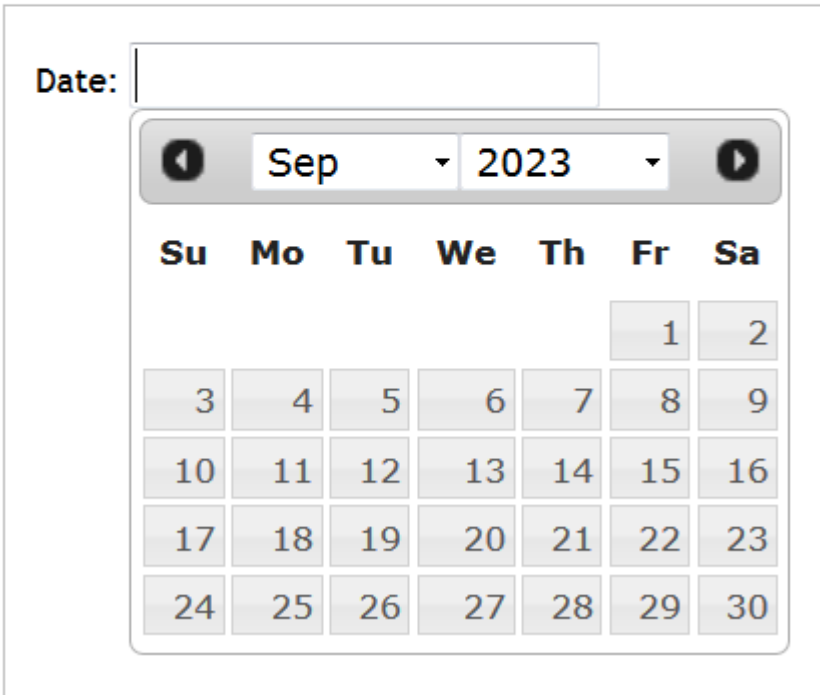
- **Toggles**: Buttons, sliders, or other widgets that allow on/off functionality. The following diagram shows an example of it:



-

- **Date fields**: The following screenshot shows a date field:



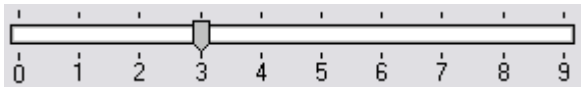


- **Navigational tools**:
  - **Breadcrumbs**: Graphical navigation aid that shows a user where they are currently located within a directory structure. Commonly seen on websites and in filesystem tools, such as Windows File Explorer.

Attribution: User:EnEdC [GPLv2 (https://www.gnu.org/licenses/old-licenses/gpl-2.0.html)], from Wikimedia Commons

-

- **Sliders**: Allows the user to move an indicator along a line to select a value; often allows the user to click on a position to set the indicator:



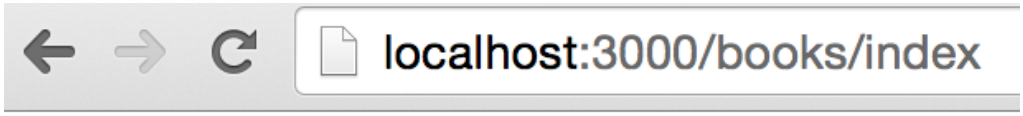
-

- **Search fields**: Normally a single-line textbox that allows the user to enter text to search for within a filesystem, database, and so on. Real-time results can populate a drop-down list, such as Google Search or Wikipedia:

Attribution: screenshot by DTankersley (WMF) [CC BY-SA 3.0 (`https://creativecommons.org/licenses/by-sa/3.0`) or GFDL (`http://www.gnu.org/copyleft/fdl.html`)], through Wikimedia Commons

-

- **Pagination**: Used to inform the user how many pages are present within a document. Normally indicated by a page number on printed documents, electronic documents may show how many pages are present overall, as well as allowing one or more ways to move between pages:



Books

Lipsum Book 1

Some Author

Lipsum Book 2

Some Author

Lipsum Book 3

Some Author

Lipsum Book 4

Some Author

Lipsum Book 5

Some Author

---

1 2 3 4 5 … Next › Last »
Displaying books **1 - 5** of **50** in total

Attribution: Moharnab Saikia [CC BY-SA 4.0 (`https://creativecommons.org/licenses/by-sa/4.0`)], from Wikimedia Commons

-

- **Tags**: Metadata comprised of keywords or terms associated with a particular piece of information, often selected by a file uploader, site administrator, users, and so on:

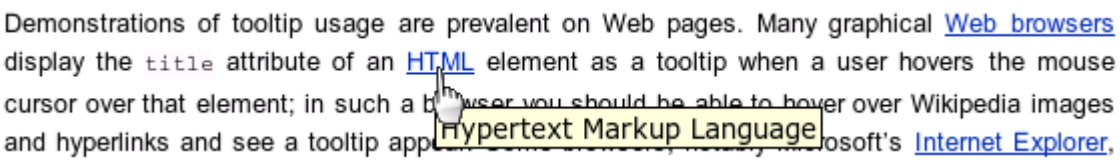
Attribution: Original by Markus Angermeier; vectorized and linked version by Luca Cremonini [CC BY-SA 2.5 (`https://creativecommons.org/licenses/by-sa/2.5`)], through Wikimedia Commons

-

- **Icons**: Graphical images used to supplement textual information. They can provide a visual representation of an operation, links to files or web pages, activate programs, and so on:

- **Informational tools**:

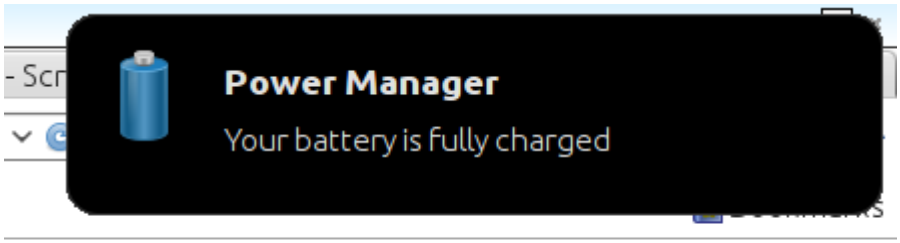
- **Tooltips**: When hovering over an item, such as a hyperlink, a small pop-up window appears with more information about the item:


Demonstrations of tooltip usage are prevalent on Web pages. Many graphical Web browsers
display the title attribute of an HTML element as a tooltip when a user hovers the mouse
cursor over that element; in such a b
er over Wikipedia images
and hyperlinks and see a tooltip app
osoft's Internet Explorer,
Hypertext Markup Language

- **Progress bars**: Visual representation of the completion progress for a computer operation, such as downloading or uploading a file, installing software, and so on:

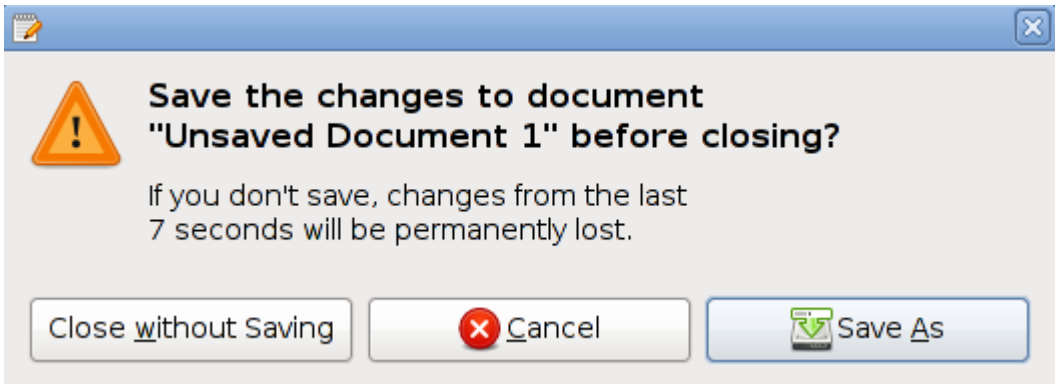
Attribution: Simeon87 [GPL (`http://www.gnu.org/licenses/gpl.html`)], through Wikimedia Commons

- **Pop-up notifications**: Graphical communication tool that displays information without forcing the user to immediately respond to the notice:



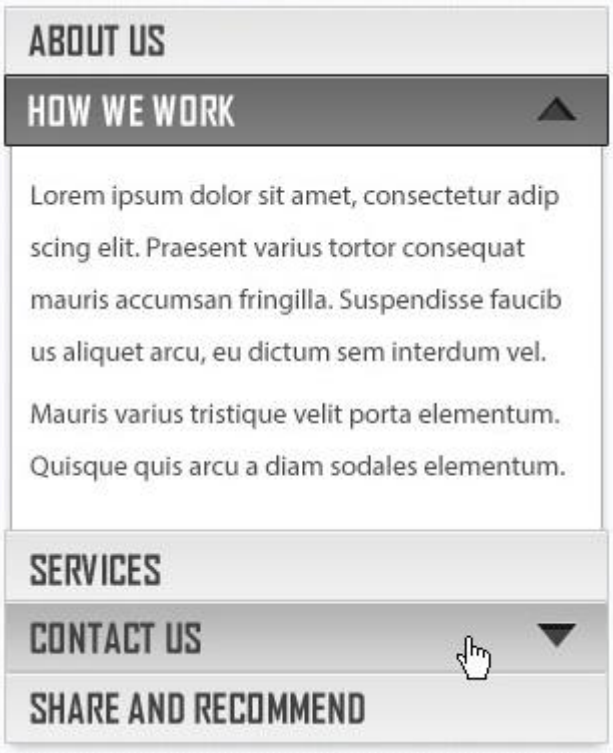
Attribution: Mudd1 [GFDL (`http://www.gnu.org/copyleft/fdl.html`) or CC BY-SA 3.0 (`https://creativecommons.org/licenses/by-sa/3.0`)], through Wikimedia Commons

- **Message boxes**: Also called dialog boxes, these are small windows to display information to the user and require a response before disappearing. Boxes can be modal (requiring user action before any other action on the system can be performed) or non-modal (the user can still use the system without responding to the box):


Save the changes to document
"Unsaved Document 1" before closing?
If you don't save, changes from the last
7 seconds will be permanently lost.
Close without Saving
Cancel
Save As

Attribution: Bruce89 at en.Wikipedia [GPL (`http://www.gnu.org/licenses/gpl.html`], through Wikimedia Commons

- **Containers:**
  - **Accordion**: A set of vertically-stacked items list. Clicking on an item expands the information content for that item:


ABOUT US
HOW WE WORK
Lorem ipsum dolor sit amet, consectetur adip
scing elit. Praesent varius tortor consequat
mauris accumsan fringilla. Suspendisse faucib
us aliquet arcu, eu dictum sem interdum vel.
Mauris varius tristique velit porta elementum.
Quisque quis arcu a diam sodales elementum.
SERVICES
CONTACT US
SHARE AND RECOMMEND

Attribution: `http://melaychie.deviantart.com/` [CC BY 3.0 (`https://creativecommons.org/licenses/by/3.0`)], via Wikimedia Commons

  -

  - **Tabs**: Displays the titles of multiple windows within a single pane, like tabs in a filing cabinet. Each tab has its own information, so selecting different tabs displays different data:

Attribution: Maria Sieglinda von Nudeldorf [CC BY-SA 3.0 (`https://creativecommons.org/licenses/by-sa/3.0`)], from Wikimedia Commons

Best practices

While designing an interface, the key factor is understanding the expected user. Is your application a general-purpose program that could be used by anyone in the population? Or, is it targeted for a specific category of users, such as doctors, who will be expected to know certain things about the program?

The user case can help determine how to best layout a GUI, what information to include on different screens, and even what type of interaction device will be used, such as a mouse pointer versus a fingertip. In the case of the doctor example, we may decide that the doctors will use the GUI on a tablet, so their fingers will be the main way to interact with the interface. Thus, buttons should be larger and spread apart more than a desktop computer, so their fingers can easily touch them.

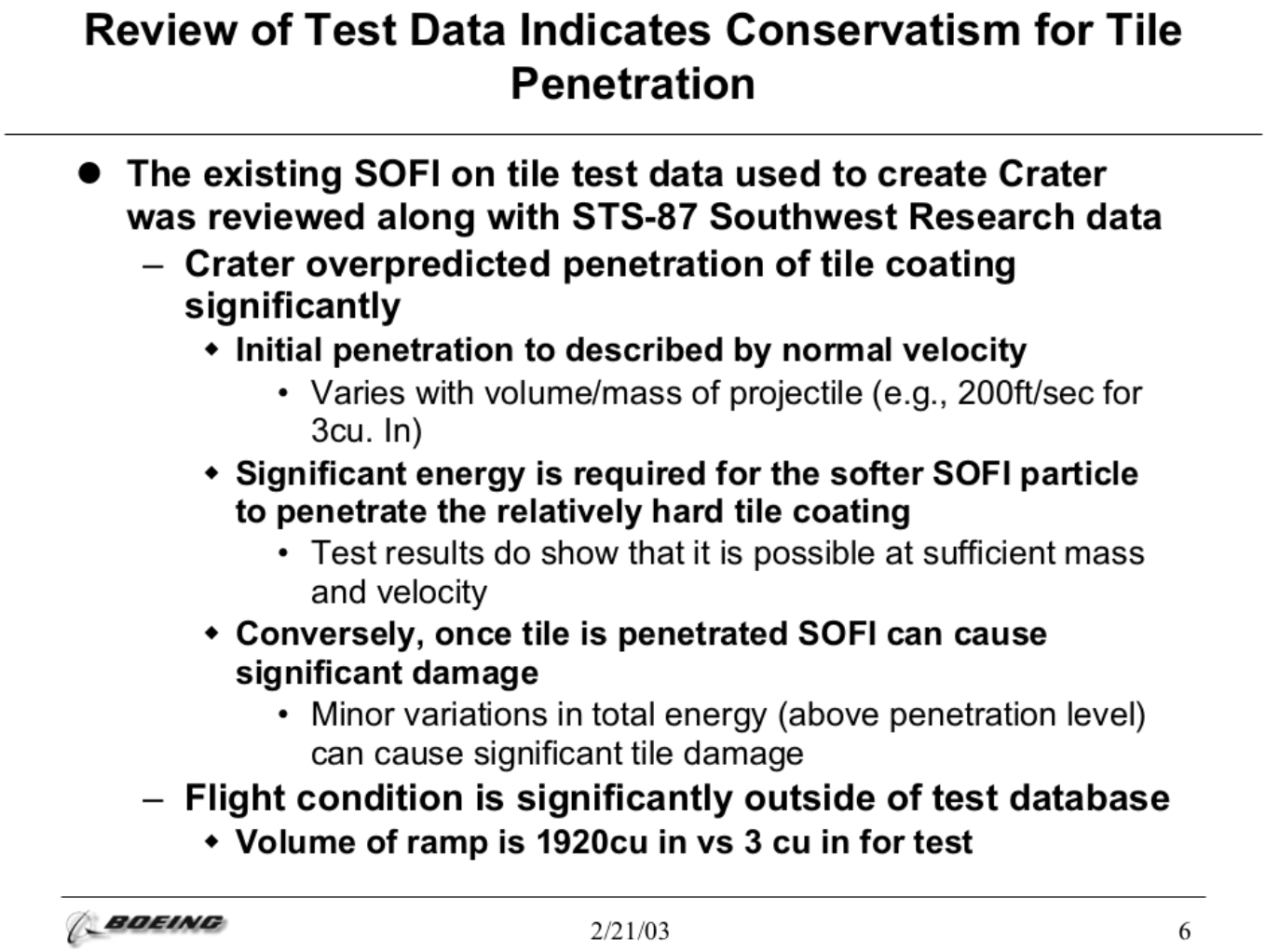
In addition, a patient's information should probably be displayed on the home page, and reference material or other, non-critical information on secondary pages. Of course, it is recommended to talk to doctors first, and during development, to ensure that the final product meets their needs.

Regardless of the user, the following are some common design criteria to consider:

- Keep the interface simple. Avoid unnecessary elements and use clear and concise words in labels and messages.
- Use a common theme. Whenever possible, use the same widgets as the underlying OS, so the user doesn't have to guess what a particular widget does. For custom schemes, ensure a particular widget does the same thing every time, regardless of where it is implemented. For example, a clock icon should only pop up a calendar, or be used to insert the current date/time, but not allow different actions while using the same icon. Having different actions means the user never knows exactly what to expect when clicking on the icon.
- Consider spatial relationships. Like newspapers, non-information helps in getting the message across; in other words, white space is important. Don't clutter the interface with too much information, but also don't spread data needlessly around the interface. Consider where the most important information should be placed, and learn where people are apt to look first and where their eyes will scan to.
- Keep the user up to date. Notify the user whenever important events occur, such as saving documents, errors, and so on. Ensure that the user knows where they are within the interface and, at a minimum, how to get to the "home page". Provide tooltips to assist in learning about elements and, when using pop-up windows, determine whether the information is important enough to justify a modal dialog box.
- Consider default values. Anticipate the most common actions of users, and create default settings that account for them. If there are settings and configurations required for your application, attempt to preconfigure them so that a user could just accept the defaults and have an acceptable output.
- Use color and texture to aid in information retrieval. Not only color but also contrast and texture can be used to highlight information. There are a number of websites and books available that discuss the making of a color-blind-friendly interface; in addition to helping color-blind users, these changes can help others by quickly showcasing different information. Also, be wary of the trend toward minimalist, low-contrast GUIs that have minimal contrast between colors (like varying shades of gray) and rely on sans serif fonts, which can be hard for some people to read. The Nielson-Norman Group (`https://www.nngroup.com/articles/low-contrast/`) provides a good example and explanation of this.
- Use typography to showcase important information. Following on from the previous bullet, the type and size of the font used can clarify the information. For example, the following screenshot is a slide from a NASA report about the Space Shuttle Columbia explosion. What information is most important? In the following screenshot, some of the most critical information, such as Test results do show that it is possible at sufficient mass and velocity as well as Volume of ramp is 1920cu in vs 3 cu in for test, are not

highlighted in any manner, even though they are probably the most critical elements of the slide. They have the same, or less, priority of information on the slide as any other bullet point.

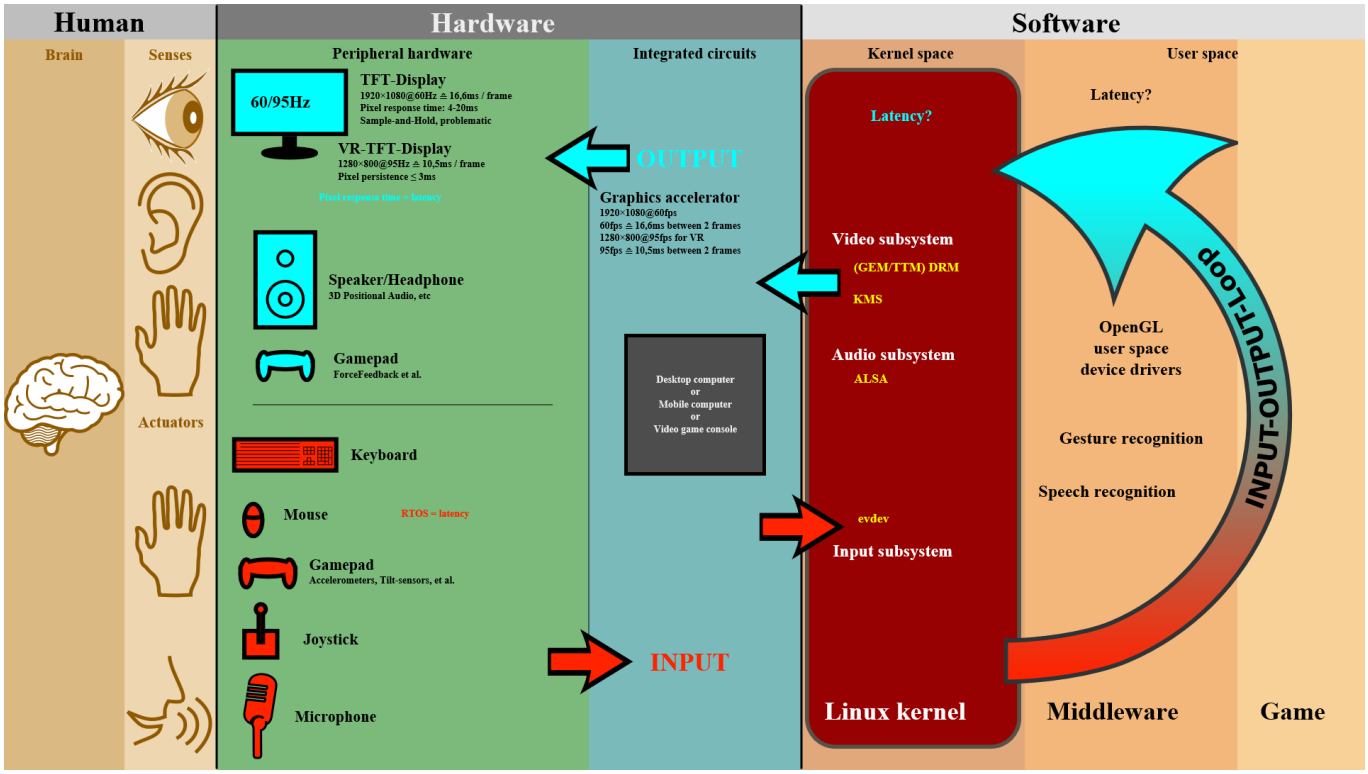
The following screenshot is an example of identifying importance:


Review of Test Data Indicates Conservatism for Tile Penetration
The existing SOFI on tile test data used to create Crater was reviewed along with STS-87 Southwest Research data
Crater overpredicted penetration of tile coating significantly
Initial penetration to described by normal velocity
Varies with volume/mass of projectile (e.g., 200ft/sec for 3cu. In)
Significant energy is required for the softer SOFI particle to penetrate the relatively hard tile coating
Test results do show that it is possible at sufficient mass and velocity
Conversely, once tile is penetrated SOFI can cause significant damage
Minor variations in total energy (above penetration level) can cause significant tile damage
Flight condition is significantly outside of test database
Volume of ramp is 1920cu in vs 3 cu in for test
BOEING
2/21/03
6

Identifying importance

User environment

The user environment comprises the traditional five senses, as determined by the computer. The computer programmer is responsible for how the user will interact with the computer. The following image displays how the user interaction cycle works:



Human-computer interaction

CC BY-SA 3.0 Shmuel Csaba Otto Traian

The preceding image shows how the user provides input to a computer through a particular hardware device. The computer hardware converts the input information to software commands that are processed by the operating system, and the appropriate application.

Once the data has been processed, the process returns to the user: the application (or a different one, if appropriate) generates a return signal that is processed by the OS. The return signal is converted into the correct output signal, such as video or audio, and that signal is sent to the appropriate hardware device, where the output signal is finally received and interpreted by the user. Then the user responds to that signal and provides a new input signal, and the whole process starts anew.

The user's environment has to be accounted for, to make an appropriate interface for the situation. Enough information has to be provided without overwhelming the user with information overload, as well as "noise": information that is irrelevant to the task at hand. The following are a couple of examples of how designing a user interface with no regard to the user's environment can affect the situation:

- **Three Mile Island**: In 1979, the reactor at the Three Mile Island power plant had a stuck relief valve. However, the control panel's indicator light only indicated if the open/close signal was sent to the valve, not the true position. So, the relief valve was stuck open; but the control panel was indicating that it was

closed. Hence, the reactor coolant leaked out to the point that the reactor overheated, which then released radioactive gas into the air outside the power plant.

- **USS Vincennes**: In 1988, the US Navy cruiser USS Vincennes shot down a civilian airliner after mistaking it for an enemy plane approaching on an attack run. The problem was that the radar operator's console on the ship had three large screens that showed the airspace around the ship, but none of them showed a plane's speed, range, or altitude. That information could only be shown by clicking on it with a cursor, and that information was provided by a separate, smaller screen.In addition, there were two separate cursors to highlight the desired target. One would track an item on the big screens, but a separate one had to be used to get the plane's flight information. A crucial fault, though, was that none of the provided information indicated how quickly a plane was gaining or losing altitude.Thus, the operator had the desired target highlighted on the big screen with the first cursor, but the second information cursor was on a different military jet. Thus, the information displayed on the smaller screen was of an actual military jet, while the desired target was a civilian plane.

While the preceding two examples aren't directly related to a normal GUI, they do show that the environment in which the user will be expected to use an interface plays a large factor in how it should be designed. Normally, people will interact with GUIs in a stress-free environment but, for a programmer or designer, the worst-case scenario should be expected.

When it comes to the GUI for our fuel farm scenario, it is advised to make it simple to use and understand. The worst-case situation is that the user needs to deal with a fuel spill and, thus, stop flow quickly to prevent a possible explosion.

Graphical frameworks

There are a number of GUI frameworks available nowadays. Since we are using Python, we will look at Python-specific frameworks:

- **Tkinter**: This is actually a Python binding for the Tk GUI toolkit. It is considered to be the standard Python GUI framework that is available on all OS installations of Python. For a long time, the widgets included in Tkinter didn't use the OS-scheme, so Tkinter applications looked out of place. That has been fixed in the latest versions, so now Tkinter programs look like native applications.
- **wxPython**: This uses a Python wrapper for the cross-platform wxWidgets toolkit. There was a time when it was considered as the replacement for the built-in Tkinter framework, but that hasn't happened. The main code is compatible with Python 2.x, while the Phoenix Project was designed to create a Python 3.x-compatible version from the ground up. wxPython uses native OS widgets, so it looks like a native application. A graphical designer, wxGlade, is available to help lay out the GUI prior to coding the functionality.
- **PyQT**: Yet another framework that wraps a Python binding around another toolkit, in this case the cross-platform Qt toolkit. PyQT is considered by many to be the primary GUI framework for Python applications, as Qt was designed for business application development and has more than 400 classes and over 6,000 functions and methods. Of course, with that power comes a steeper learning curve than other toolkits.
- **Kivy**: Unlike the other tools, Kivy is a pure-Python library and not a wrapper around another language's toolkit. It is cross-platform, as well as cross-device, so you can run the same application on a Windows PC, an iPad, or Android phone. Programs can be written in pure Python, or optionally you can create a hybrid

program that uses Python for functionality and the Kv language for user-interface markup; this helps separate functionality from design.

There are many other GUI frameworks available, so the reader is encouraged to look at the options available before committing to a particular one.

Summary

In this chapter, we talked about how to make a graphical user interface actually functional for the user along with some best practices. We discussed the user environment and how it affects the design practices for an application, and, at the end, we discussed some of the most common Python GUI frameworks available for developers.

In the next chapter, we will actually create a GUI for the fuel farm project and ensure that it provides the functionality we want.

Creating a Graphical User Interface

While we covered a number of GUI frameworks in the last chapter, we will use **Kivy** for this particular development project. It is pure Python, so you don't have to concern yourself with trying to configure a separate environment; it is cross-platform as well as cross-device, so Kivy applications have a wider user base; and it is in active development, with new releases coming out every few months, in addition to a steadily improving library of tools.

In this chapter, we will cover the following topics:

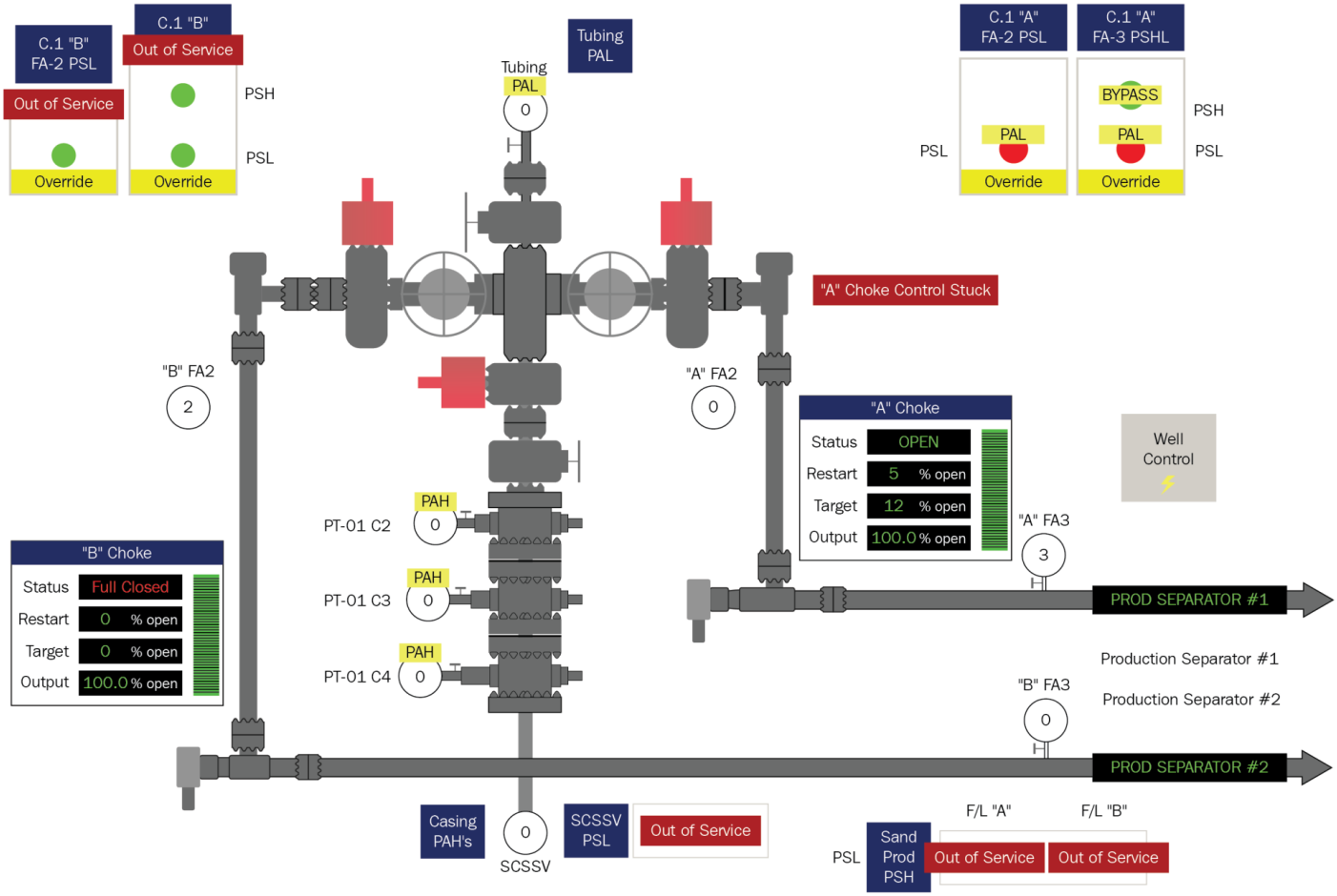
- Wireframing
- Coding the interface
- GUI testing

Wireframing

When designing a GUI, it is common to sketch out what the final interface should look like. This is known as wireframing. While there are applications designed explicitly for wireframing, even something as simple as MS PowerPoint can be used.

Ideally, we would talk to the end users and figure out what they wanted in a GUI, allowing them to test each version of the software until we found the best solution. However, when it comes to a **Human-Machine Interface** (**HMI**), as GUIs for industrial applications are commonly called, the interface is pretty much dictated by the system being used.

HMIs have to show what the system looks like, much like a schematic diagram, as well as telling the operator the current system parameters and conditions. The following screenshot shows a representative example of an HMI for an industrial application:



Example HMI

Frequently, the HMI allows the operator to directly control the system by manipulating GUI elements, such as buttons or switches. The HMI is remotely connected to the actual components, much like we simulated in the fuel farm program in `Chapter 9`, *Writing the Fueling Scenario*.

Because HMIs are basically system drawings, we don't have much to wireframe for this project. We can actually use the schematic drawing itself, as all of the necessary information is already shown on the schematic drawing.

Coding the interface

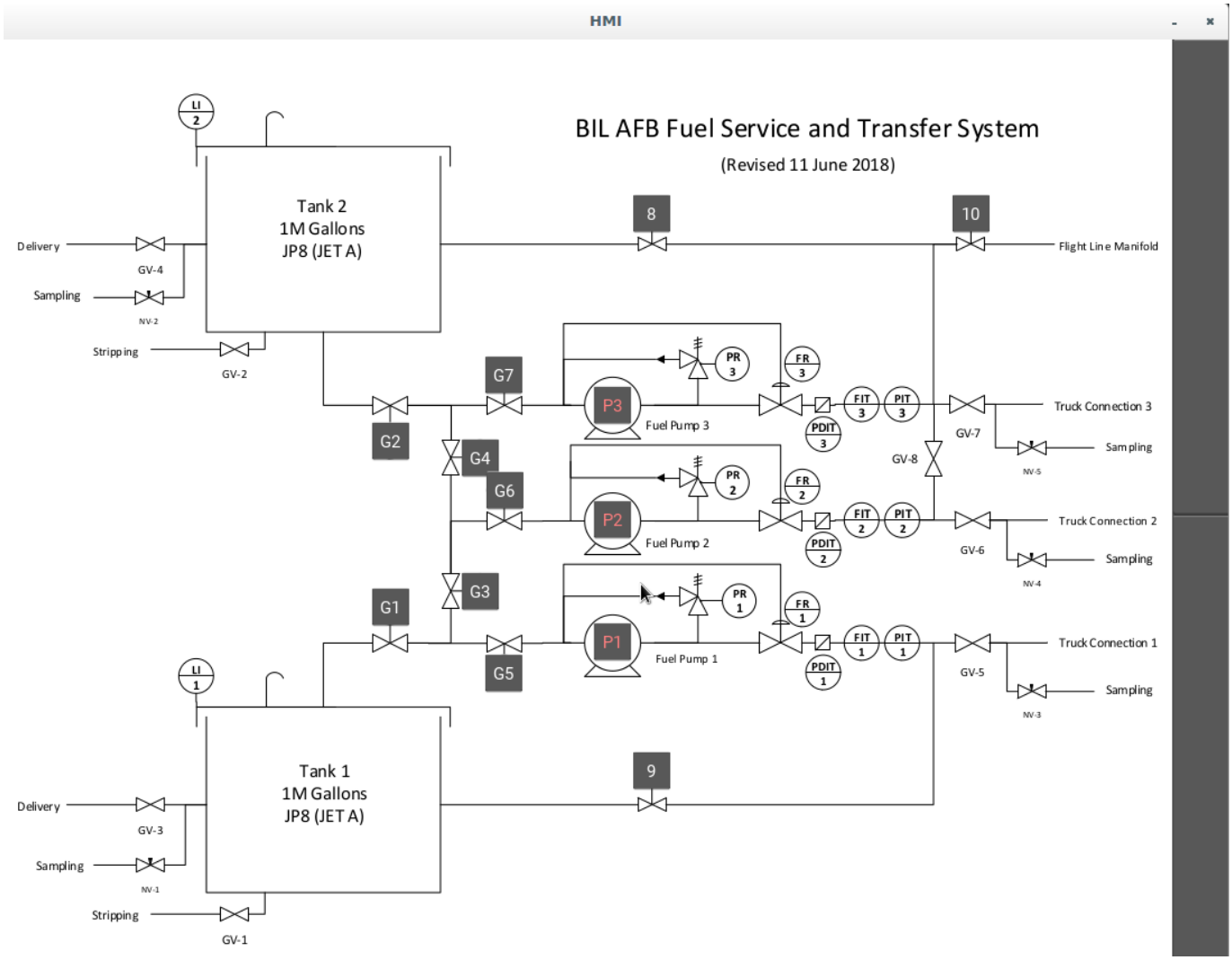
As we will be using Kivy, you can go to `https://kivy.org/` and review the installation instructions.Depending on your OS, and whether you're using a virtual Python environment, Łand other factors, there are different ways to install the software.

By default, Kivy applications are designed to scale according to the device or the user's desires. If we were to allow this, we would have to hand-draw the schematic drawing within Kivy, or otherwise create the HMI from scratch. If we used the schematic drawing as a background image and placed widgets on top, the widgets would not maintain the same position relative to the schematic drawing when scaled.

Having a scalable image is the most desirable option, but for the purpose of this book, we will use the schematic drawing as a background image and block scaling. Therefore, any widgets that we place on top of the schematic drawing won't have to worry about the drawing moving underneath them.

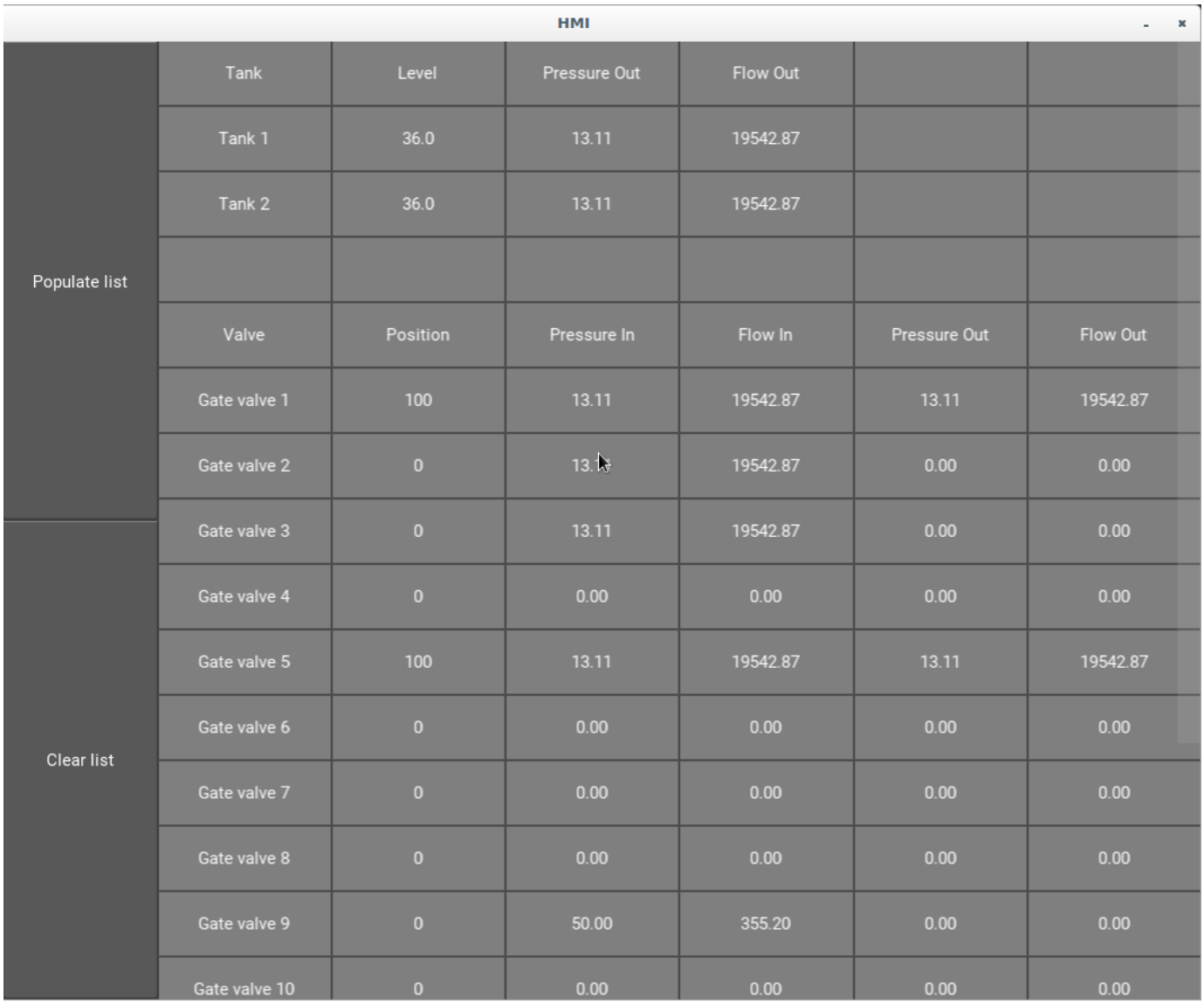
As this is not a Kivy-specific book, we will only cover the fundamental Kivy processes necessary to code the fuel farm GUI. We will talk about two specific files in this chapter: `hmi.kv`and `hmilayout.py`. They define the widget layout and program functionality, respectively, and allow the programmer to separate logic from presentation. Frequently, the two files are coded side by side, with logic leading to widget layout and vice versa. To make it easier for the reader, we will talk about the logic file first, and then we'll cover the layout.

To give the reader an idea of what the expected outcome is, the schematic portion of the HMI is shown in the following diagram:



Kivy HMI (part 1)

The parameter data table is shown in following table:



Kivy HMI (part 2)

As Kivy is designed to be used with mobile devices, the two screens can be viewed by grabbing the side bar (dark gray in the first image) and swiping back and forth. On a computer, this is done using the mouse to swipe.

Kivy logic file

The first thing we will work on is the Python logic for our application. It is demonstrated in the multi-part listing of `hmilayout.py`, as follows:

```
# hmilayout.py (part 1)
1  import sys
2  sys.path.extend(["/home/cody/PycharmProjects/VirtualPLC"])
3
4  import Models.FuelFarm.components as components
5  import Models.FuelFarm.functionality as functionality
6
7  from kivy.app import App
8  from kivy.uix.pagelayout import PageLayout
9  from kivy.config import Config
10
11 import kivy
12 kivy.require("1.10.0")
```

In part 1, we import all of the important modules that we will need. Lines 1 and 2 show the path extension code to ensure the smooth attempt to execute the program. Lines 4 and 5 assign alias names to the `components.py` and `functionality.py` files, so we don't have to continually type the entire path every time.

Lines 7-9 import the important pieces from Kivy. There are a number of different layouts available in the `uix` module, so it is worth looking at them to see which ones might best fit your GUI.

Line 12 is important, as it dictates which Kivy version the program is written for. If a user has an older version of Kivy installed, an exception will be generated and the program won't run; this is because Kivy changes frequently. If a newer version is installed, the exception won't occur.

The following code snippet is part 2 of `hmilayout.py`:

```
# hmilayout.py (part 2)
1  # Fix the drawing to a set size and prevent scaling
2  Config.set("graphics", "width", "1112")
3  Config.set("graphics", "height", "849")
4  Config.set("graphics", "resizable", False)
5
6  class HMILayout(PageLayout):
7   # Methods are associated with their class; each class would have its own .kv file
8      @staticmethod
9      def on_state(device): # Get the status of the device
10         if device.state == "down":
11             if device.group not in ["pump1", "pump2", "pump3"]:
12                 exec("functionality.{}_open()".format(device.group)) # Dynamically
call valve open()
13             else:
14                 exec("functionality.{}_on()".format(device.group)) # Dynamically
call pump on()
```

Lines 2-5 are the commands that force the schematic drawing to not be scalable, as well as forcing the displayed image to have the same dimensions as the original file. This way, it is easy to use a program such as GIMP or Photoshop to overlay a grid and determine the rough position of where widgets should be placed.

In normal Python fashion, we create a class in line 6 to contain all the items that will create the GUI. In this case, the class inherits from the `PageLayout` class, allowing us to simply inherit the layout and functionality of widgets from the parent class. As the comment on line 7 indicates, if multiple classes are used, then each class would have its own `.kv` layout file associated with it.

We define a static method in lines 8-14 that receives the status state of a particular widget (in this case, a toggle button). A static method is used because we don't care whether a class or instance calls the method, so a static method operates more like a regular function that can be called either from an instance or a class; this is useful in a GUI, as it makes the method more universal in use. If the widget for a particular component is "down", or showing blue on the HMI, the device is either open (if a valve) or on (if a pump).

We use the Python `exec()` function to call the appropriate action, depending on the component.
The `exec()` command effectively uses the string argument passed into it as a command to the Python interpreter. This choice was made to allow the string command to dynamically determine the name of the component; this is such as having a large `if...else` block, where each command would be the same but only the component name was different.

The following code snippet is part 3 of `hmilayout.py`:

```
# hmilayout.py (part 3)
1 def populate(self):
2 # Make dictionaries to populate table
3     tank_properties1 = {}
4     tank_properties2 = {}
5
6     valve_properties1 = {}
7     valve_properties2 = {}
8     valve_properties3 = {}
9     valve_properties4 = {}
10     valve_properties5 = {}
11     valve_properties6 = {}
12     valve_properties7 = {}
13
14     pump_properties1 = {}
15     pump_properties2 = {}
```

Part 3 shows a method that will populate the parameter table. This method is linked to one of the buttons on the table slide of the GUI; initially, the table is empty so the button is provided to fill it. This first listing shows all of the empty dictionaries we will use later; there are alternative ways to make dictionaries, such as the `dict()` method, but it is sometimes easier for clarity to create empty ones and populate as needed.

It should also be noted that the current code does not automatically update the table when data changes. So, if a user clicks one of the toggle buttons on the GUI, the Populate Tablebutton needs to be clicked as well to show the changes.

The following code snippet is part 4 of `hmilayout.py`:

```
# hmilayout.py(part4)
1 pump_properties3 = {}
2
3 # Convert instances to dictionaries
4 for key, value in vars(components.tank1).items():
5     tank_properties1[key] = value
```

```
6 for key, value in vars(components.tank2).items():
7     tank_properties2[key] = value
8
9 for key, value in vars(components.gate1).items():
10     valve_properties1[key] = value
11 for key, value in vars(components.gate2).items():
12     valve_properties2[key] = value
13 for key, value in vars(components.gate3).items():
14     valve_properties3[key] = value
```

Line 1 finishes the empty dictionary creation, and then we move into the expressions. The expressions convert all of the component instances we created into key:value pairs that can populate the dictionaries. Because this methodology is the same for different components, we will not cover all of the components as coded. However, the complete `hmilayout.py` file is available in this book's code repository for review.

The following code snippet is part 5 of `hmilayout.py`:

```
# hmilayout.py (part 5)
1 # Populate table
2 self.table.data = [{"value": "Tank"}, {"value": "Level"}, {"value": "Pressure Out"},
{"value": "Flow Out"},
3 {"value": ""}, {"value": ""},
4 # Tank 1
5 {"value": tank_properties1["name"]},
6 {"value": str(tank_properties1["_Tank__level"])},
7 {"value": "{:.2f}".format((tank_properties1["_Tank__tank_press"]))},
8 {"value": "{:.2f}".format((tank_properties1["flow_out"]))},
9 {"value": ""},
10 {"value": ""},
11 # Tank 2
12 {"value": tank_properties2["name"]},
13 {"value": str(tank_properties2["_Tank__level"])},
14 {"value": "{:.2f}".format((tank_properties2["_Tank__tank_press"]))},
15 {"value": "{:.2f}".format((tank_properties2["flow_out"]))},
```

After we have populated the dictionaries, we need to populate the table itself. This code block takes up nearly 100 lines, so we won't cover each line here. The main thing to get from part 5 is that the table's data is determined by a list of dictionary items. This is a part of how Kivy operates, particularly the `RecycleGridLayout` class that is used for tabular data display. (We will see that class in the next section when we talk about the Kivy layout.)

The following code snippet is part 6 of `hmilayout.py`:

```
# hmilayout.py (part 6)
1     def clear(self):
2         self.table.data = []
3
4  class HMIApp(App):
5     def build(self):
6         return HMILayout()
7
8  if __name__ == "__main__":
9     HMIApp().run()
```

After we have set up the table to be populated with data from the model, we write the method (line 1) to clear the table when the Clear List button is pressed.

We create a new class in line 4 that has only one method: `build()`. The whole purpose of this class is to build the GUI we defined in the `HMILayout()` class.

At the end (line 9), we actually run the GUI when it is called.

Kivy layout file

Now that we have the logic for how the GUI is supposed to function, we can actually write the code to place the widgets where we want on the diagram. You could do this within the Python file that we just created, but best practice is to separate code logic from presentation. For example, the following code from the official Kivy tutorial shows how a Python file could both accept a touch from a finger and create a small dot on the screen of a tablet:

```
def on_touch_down(self, touch):
    with self.canvas:
        Color(1, 1, 0)
        d = 30.
        Ellipse(pos=(touch.x - d / 2, touch.y - d / 2), size=(d, d))
```

For our purposes, we will write a special `.kv` file to hold the layout-specific information and keep the Python logic in `hmilayout.py`:

```
# hmi.kv (part 1)
1 #:kivy 1.10.0
2
3 <HMIButton@ToggleButton>:
4     size_hint: None, None
5     size: 35, 35
6
7 <Row@BoxLayout>:
8      canvas.before:
9          Color:
10             rgba: 0.5, 0.5, 0.5, 1
11         Rectangle:
12             size: self.size
13             pos: self.pos
14     value: ''
```

Line 1 has the Kivy version information to ensure that the user doesn't receive errors. Lines 3-5 define the `HMIButton` size that will be used for the GUI; it inherits from the Kivy `ToggleButton` class.

`size_hint` (line 4) accepts proportional values from 0 to 1; the values are basically the percentage length of width and height. Because the values for `size_hint` are `None`, we provide actual values through `size` (line 5) to set the button to be square. If not using a set size, then only `size_hint` needs to be used and the GUI will automatically size the widget.

Lines 7-14 determine the layout of the rows in the parameters table. `canvas.before` tells Kivy to process the items beneath the line (lines 8-13) before moving onto anything else; essentially, it defines the background of the drawing canvas, where images and widgets are placed. In this case, we are defining the color of the row and the size of the cells within the table.

`Color` (line 9) accepts the red, green, blue, and alpha channel values as proportions, from `0` to `1`.
The `size` and `pos` parameters of `Rectangle` (lines 12 and 13) don't have values assigned, so they will accept the default values from the parent class, `BoxLayout`.

Line 14 provides the option for a default value to be given. As we will fill this later, we have set it to be an empty string.

The following code snippet is part 2 of `hmi.kv`:

```
# hmi.kv (part 2)
1 Label:
2 text: root.value
3
4 <HMILayout>:
5     table: table # identify this object for reference by other objects
6     swipe_threshold: .2 # Allow page turn to occur when it has been moved 20%
7     FloatLayout:
8         # First page (HMI)
9         canvas.before:
10            Rectangle:
11                pos: self.pos
12                size: self.size
13                source: "fuel_schematic.png"
```

Lines 1 and 2 finish up the `Row` from the part 1. `Label` is the name that would be placed on the row; in line 2, the text assignment indicates that the text value will be whatever the root class at the top of the hierarchy is; in this case, it is `BoxLayout`.

Starting with line 4, we define how the buttons will be placed on the GUI. Line 5 provides a reference name for this particular object, while line 6 determines how far the user must swipe before the next screen appears (whether schematic to the table or vice versa).

Line 7 declares `FloatLayout`, which organizes widgets with proportional coordinates using `size_hint`and `pos_hint` properties. It also allows widgets to overlay one another.

Lines 9-13 dictate what Kivy is supposed to do first when processing the file. In this case, we create a rectangle with inherited size and position; then we fill it with the schematic drawing. This way, the drawing is the first thing displayed by Kivy, allowing all other widgets to lay on top of it.

The following code snippet is part 3 of `hmi.kv`:

```
# hmi.kv (part 3)
1 HMIButton:
2     id: gate1
3     text: "G1"
4     pos: 347, 325
5     group: "gate1"
6     on_state: root.on_state(self)
7
8 HMIButton:
9     id: gate2
10     text: "G2"
11     pos: 347, 473
12     group: "gate2"
13     on_state: root.on_state(self)
```

After defining the `canvas.before` items, we move into creating widgets. In part 3, we define two buttons that will be placed on the schematic drawing. Lines 1-6 create gate valve 1 on the drawing, providing the Kivy identification value, the text to be displayed on the button, the absolute position of the button (from the bottom-left corner of the drawing), the group the button is associated with, and access to the `on_state()` method that we defined previously. Lines 8-13 do the same thing for gate valve 2.

Since the rest of the buttons follow a similar pattern, we will not cover them in detail here. However, the full code is available in this book's code repository.

The following code snippet is part 4 of `hmi.kv`:

```
# hmi.kv (part 4)
1 HMIButton:
2     id: pump1
3     text: "P1"
4     pos: 545, 294
5     group: "pump1"
6     on_state: root.on_state(self)
7     color: 1, .5, .5, 1
```

The preceding code listing for a pump button is slightly different from the valve buttons. Line 7 shows a color listing, in the normal RGBA setting we saw in part 1. This color defines the text color used by the button; this was done in order to help to differentiate the buttons on the drawing. Valve buttons have default text colors, while pump buttons have pinkish text.

The following code snippet is part 5 of `hmi.kv`:

```
# hmi.kv (part 5)
1 BoxLayout:
2     # Second page (table layout)
3     canvas:
4         Color:
5             rgba: 0.3, 0.3, 0.3, 1
6         Rectangle:
7             size: self.size
8             pos: self.pos
9     table: table # identify this object for reference by other objects
10    orientation: 'horizontal' # place the following layouts side-by-side
11    BoxLayout:
12        orientation: "vertical" # stack buttons
13        size_hint_x: .15 # buttons should be 15% window width
```

After we provide the code for all of the buttons on the GUI, we move onto the parameters table page. This is the page that is swiped in from the side.

Line 1 tells Kivy that we will use `BoxLayout` this time, which organizes widgets into a row or a column, depending on the orientation provided.

We define the canvas colors, rectangle size, and position in lines 3-8. Note that we didn't use `canvas.before` in line 3; with just `canvas`, items are allowed to overlap. There is also `canvas.after`, which places widgets after everything else that has been placed. You can think of the different `canvas`options as the tools that the drawing applications have for "move to back", "move to front", and so on.

In line 9, we again identify the object for future reference and line 10 tells `BoxLayout` that we will be using the horizontal orientation, to create a row of widgets.

Lines 11-13 create a new `BoxLayout` within the parent box container. This new layout will stack two buttons vertically (the Populate List and Clear List buttons).

The following code snippet is part 6 of `hmi.kv`:

```
# hmi.kv (part 6)
1         Button:
2             text: 'Populate list'
3             on_press: root.populate()
4         Button:
5             text: 'Clear list'
6             on_press: root.clear()
7     RecycleView:
8         # reference to data table
9         id: table
10        scroll_type: ['bars', 'content'] # table is scrolled by using scroll bars or
touching content directly
11        bar_width: dp(20) # sets scroll bar width to 20 px
12        viewclass: 'Row' # Refer back to Row@BoxLayout for grid appearance
13        RecycleGridLayout:
14            size_hint: 1, None # Force scrolling
15            cols: 6
```

Lines 1-6 create the buttons we talked about in part 5 and associate them to their respective `hmilayout.py` methods.

Line 7 starts a way to view datasets. `RecycleView` is the current way to create tables within Kivy; it was added in version 1.10 and replaced the old `ListView` class.

Line 9 provides the ID reference for the view; the name "table" here is the same table listed in part 2 and part 5.

Line 10 indicates that scroll bars are available for mouse users, but mobile device users can touch the table directly to scroll. Line 11 provides the width of the scroll bars.

Line 12 calls back to the `Row` layout to determine how to display the table.

Lines 13-15 actually create the layout that will be used to display information. Line 14 forces the scroll bars to appear, rather than only showing up when scrolling starts. This allows the user to easily "grab" the scroll bars for quick scrolling. Line 15 states that six columns will be used for the table.

The following code snippet is part 7 of `hmi.kv`:

```
# hmi.kv (part 7)
1             default_size: None, dp(56) # fit table to window
2             default_size_hint: 1, None # fill width
3             height: self.minimum_height # fill height
4             spacing: dp(2) # spacing between cells
```

We finish up `RecycleGridLayout` by providing for the default sizes of the table and the cells.

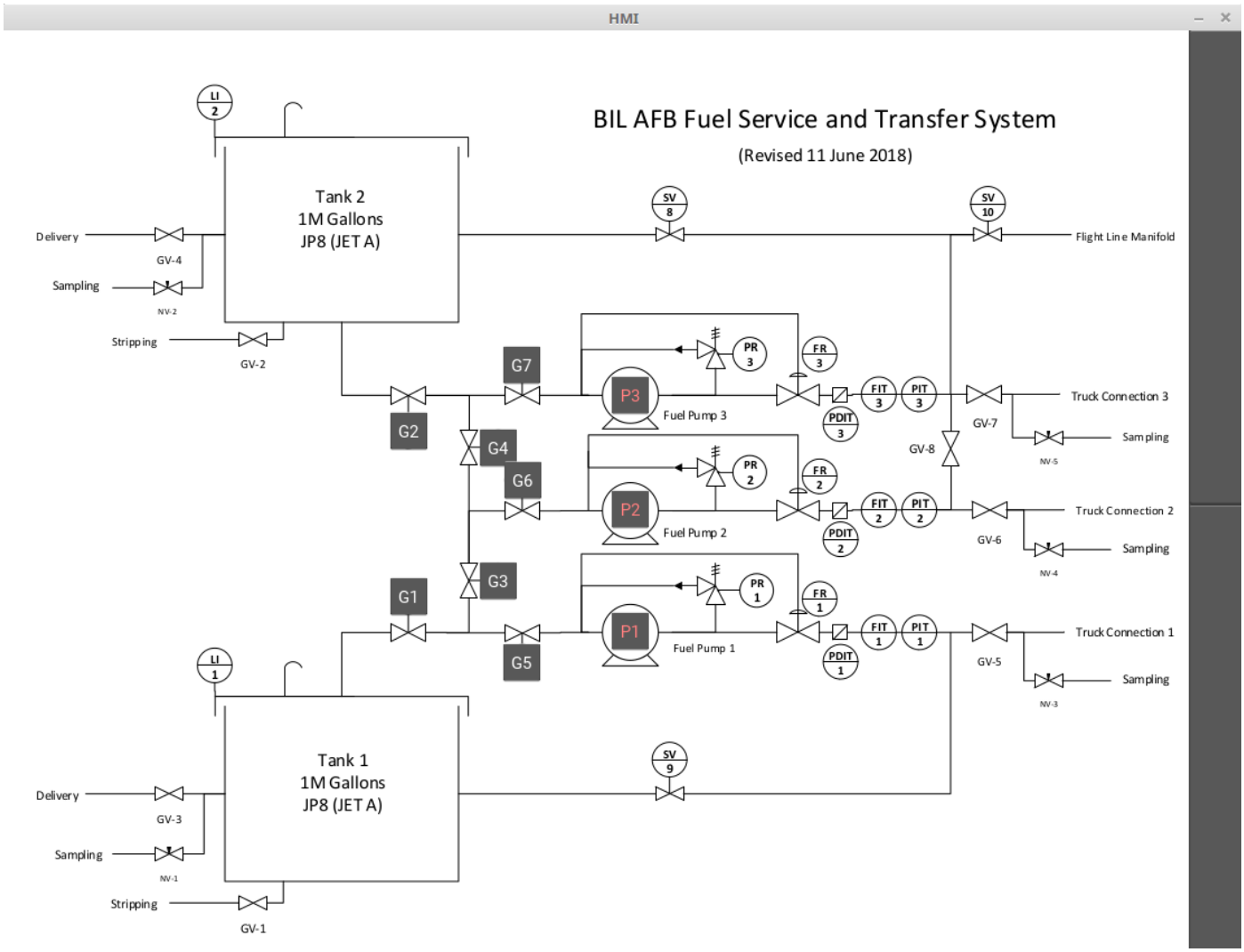
GUI testing

While there is documentation for Kivy testing, it requires the installation of `nose` (an alternative testing library, similar to `pytest`) for unit testing and `coverage` to check how much code was tested. However, the documentation doesn't seem to have been updated within the last year, as it refers to the original `nose` program, which has been deprecated in favor of `nose2`.

You can use regular, non-graphical unit tests in your code. But, if you want to test the GUI itself, you have to set up the environment for that. We won't bother with automated testing for this project, though it is advised to do that for large-scale projects. Any new versions of the fuel farm scenario would include automated tests; but, for now, we will manually test it because it is actually pretty easy in this small a project.

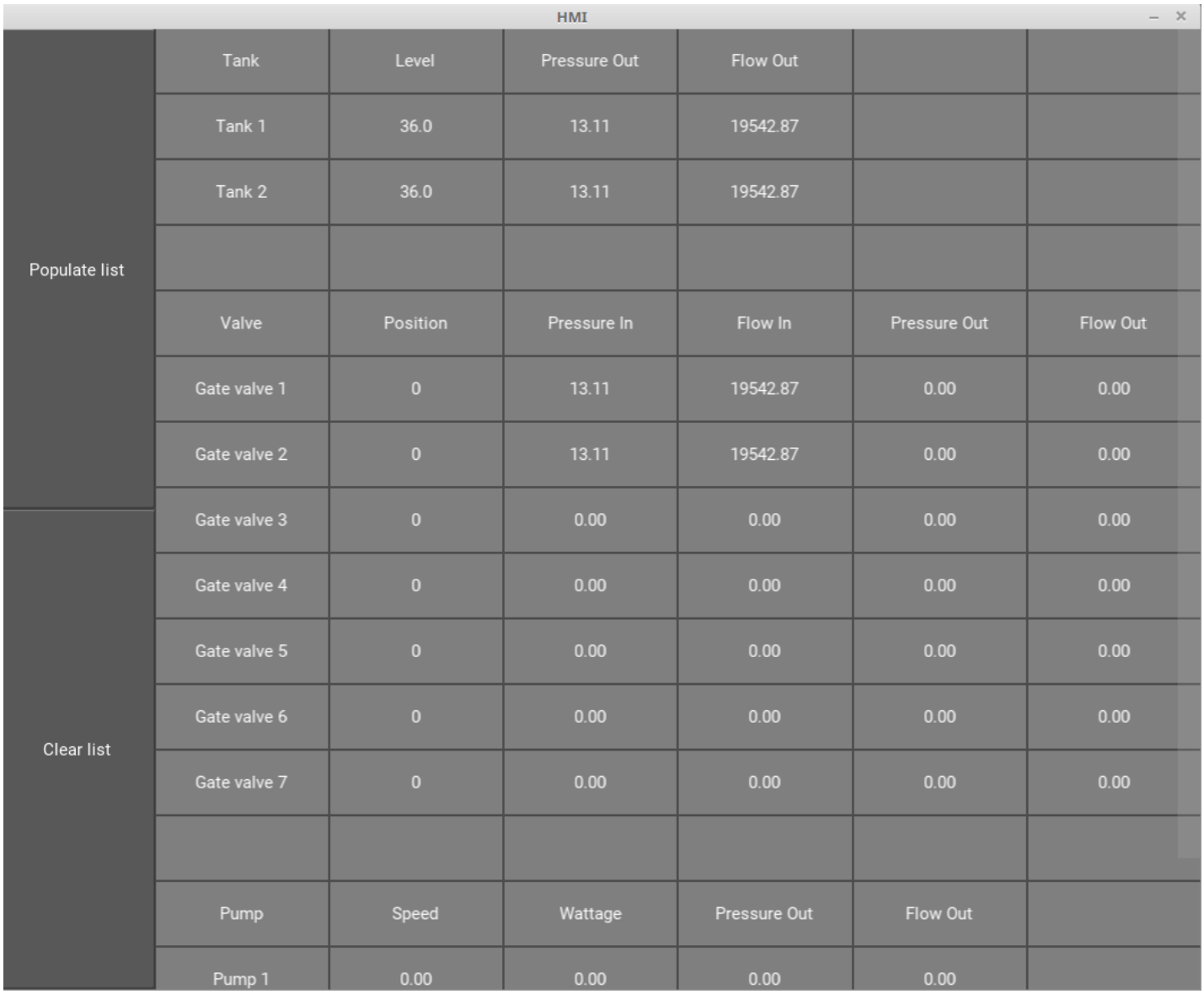
First, we just want to ensure that, when everything is closed or turned off, we should have no flow through the system. In this case, the only pressure is from the inlet of valves 1 and 2 from the static tank pressure.

The diagram of this lineup is shown in following:



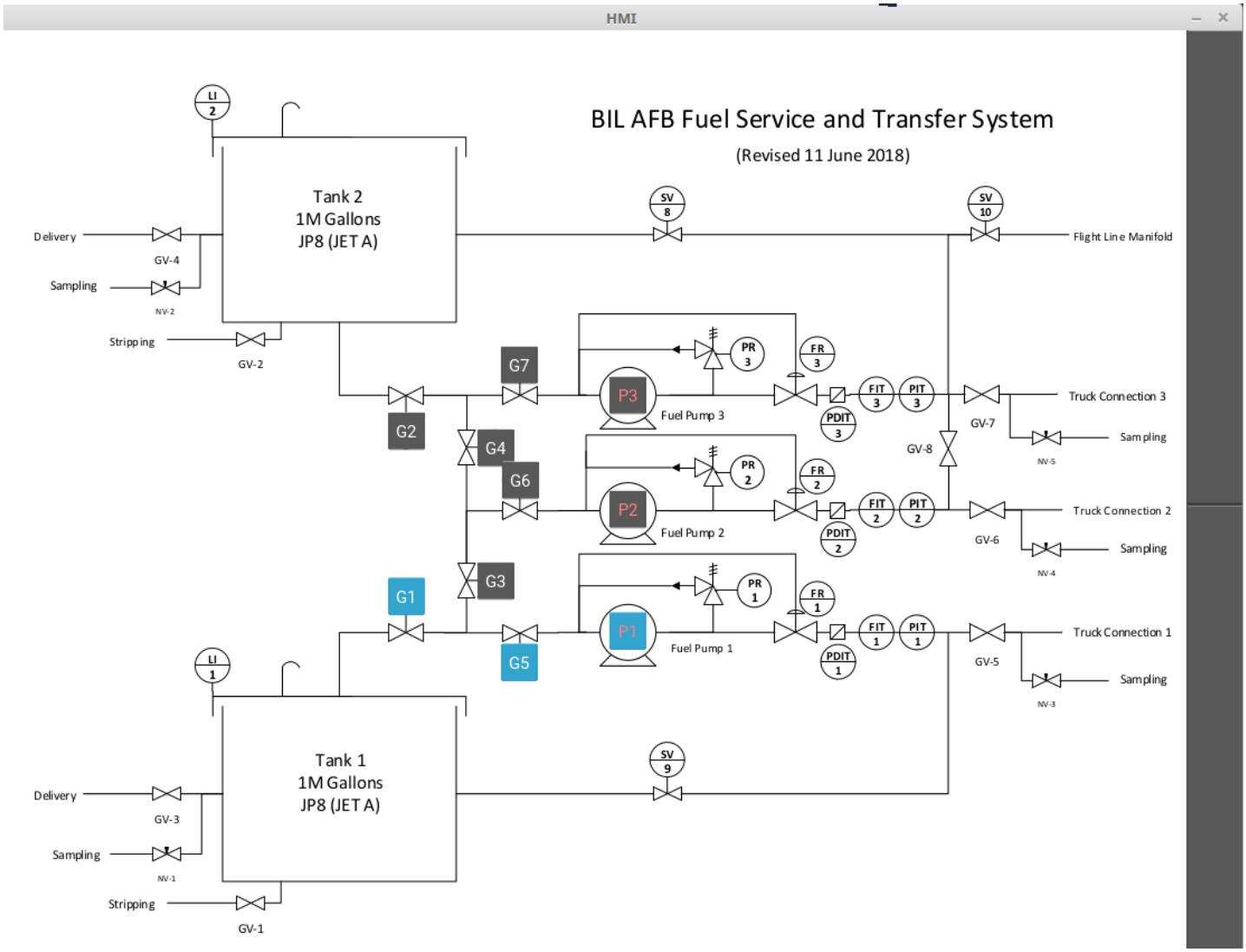
Closed diagram

The table output of the lineup diagram shown is as follows:



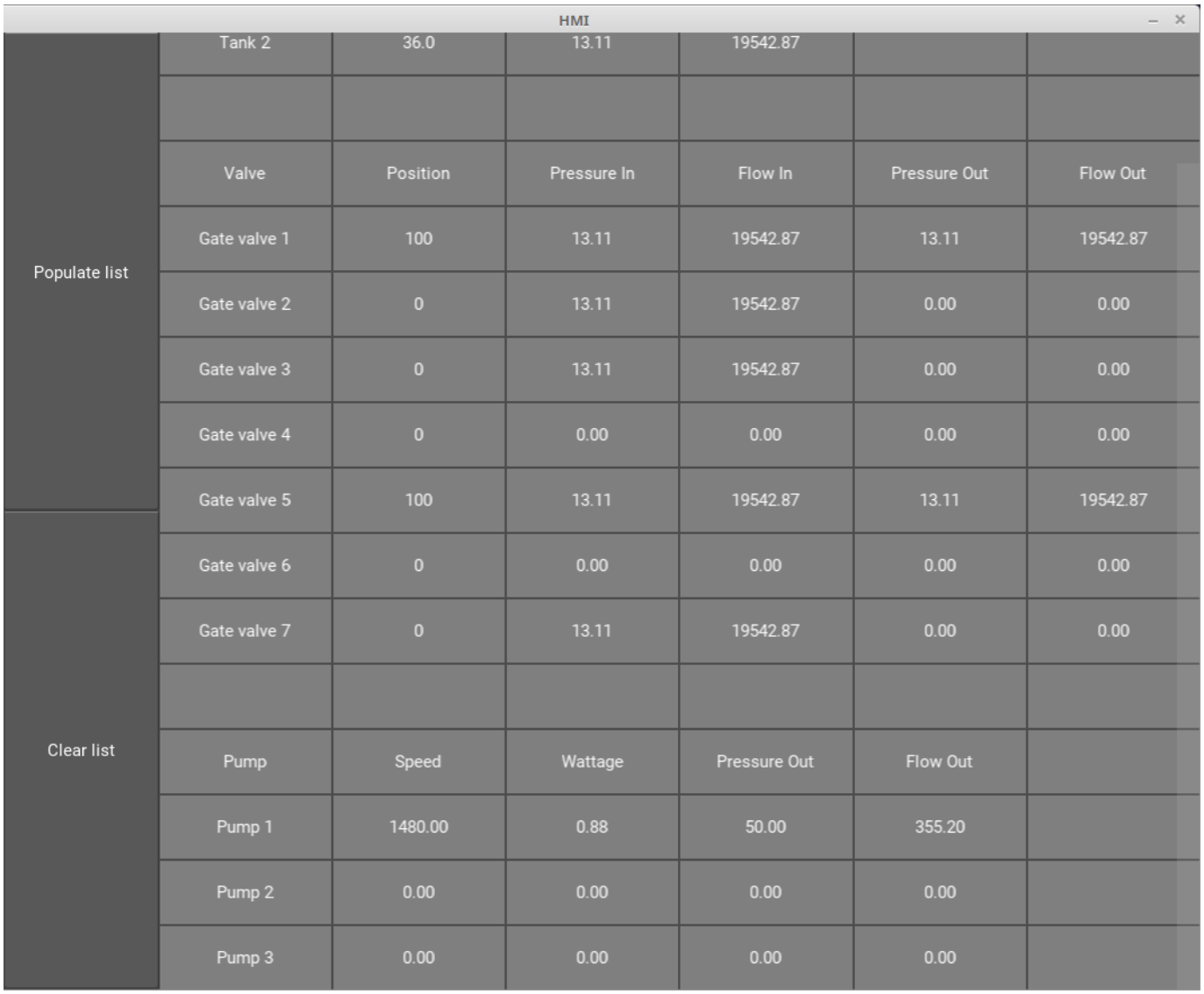
Closed table

For a simple system lineup, coming from **Tank 1** through **Pump 1**, we click the appropriate buttons, as shown in the following diagram:



Tank 1-Pump1 drawing

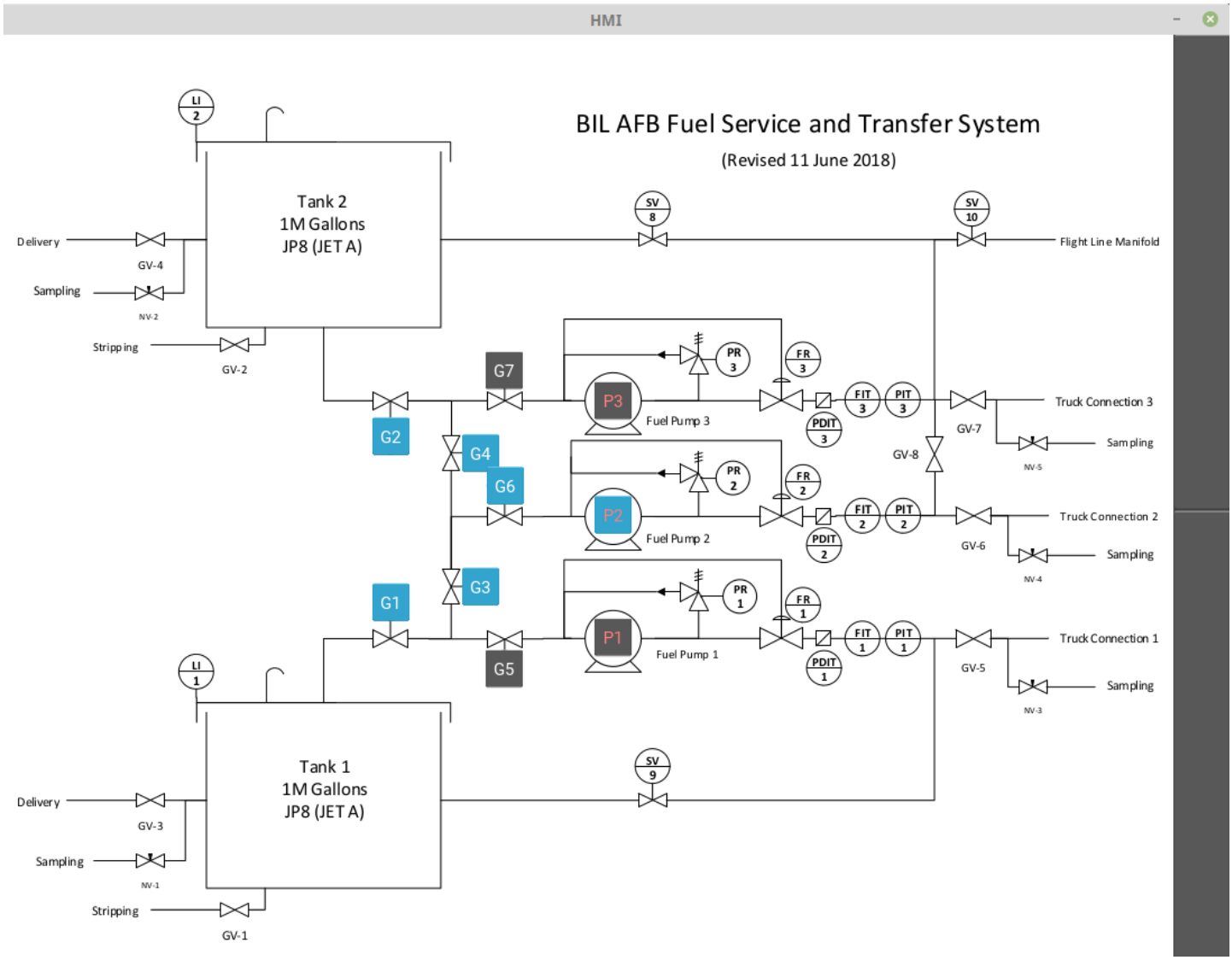
The output after clicking buttons in a simple system lineup that is coming from **Tank 1**through **Pump 1** is provided in the table shown in the following:



Tank 1-Pump 1 table

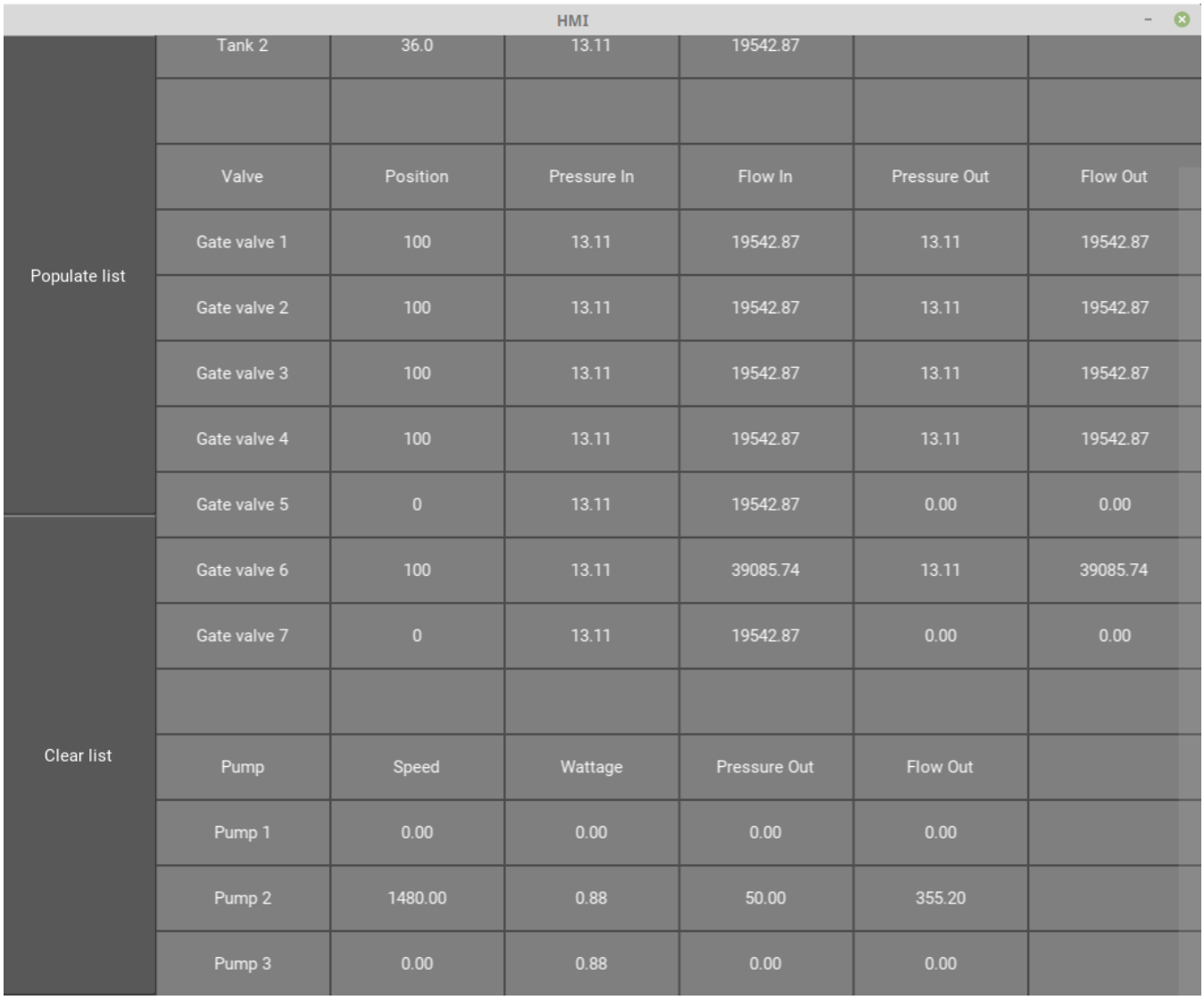
As shown in preceding the screenshot, we now have pressure and flow out of valves 1 and 5, and flow out of **Pump 1**. You can double-check the values that are shown; they should match your calculations as well as being part of your unit tests for the text-based program.

Another test is to ensure that the flow from both the tanks is represented correctly. The lineup is shown in the following diagram:



Both tanks to pump 2

The output for the preceding lineup is shown in the following:



Both tanks to pump 2 table

As shown in the preceding screenshot, the flow through the valves is correct, with valve 6 receiving the combined flow from both tanks, while the pressure out is the same as the other valves (static pressure in the tanks is the only pressure source). The pump information is also correct, even though we have the combined flow coming into it; it can only pump so much, so it doesn't matter what the flow rate coming into it is.

Summary

In this chapter, we learned about wireframings and how they are used to mock up graphical interfaces. We also saw that, sometimes, the wireframe may already be available to you, such as a schematic diagram, so you don't have to make a new one or you can use it directly as the GUI itself.

We saw how to use Kivy's API calls to create a simple GUI using a pre-made schematic drawing and how to generate an output table of information. After making the GUI, we learned how to map the original text-based program to the GUI widgets to provide the user with a point-and-click method to adjust the program.

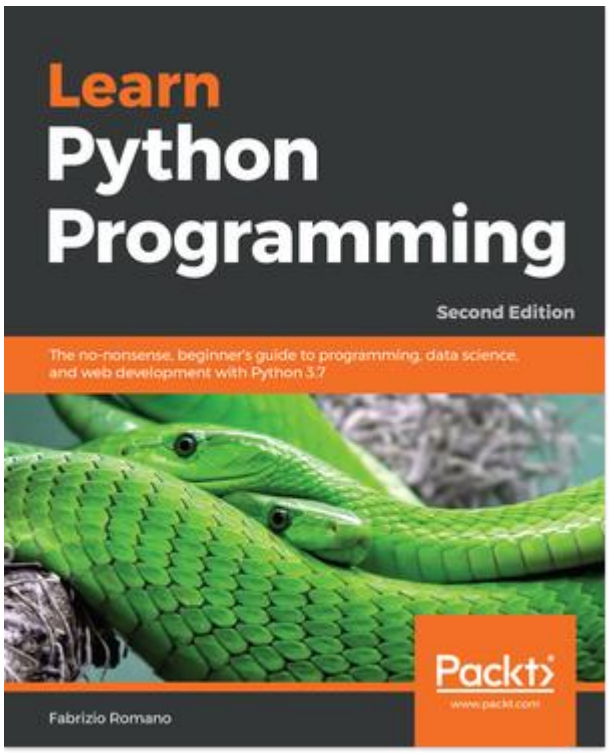
Finally, we looked at how to manually check the values of the fuel farm project and saw that, sometimes, it may be easier to manually test a GUI than write the automated tests if the GUI is simple enough.

There is a lot of information in this fuel farm simulation project, but it should suffice to demonstrate what can be done with Python. While actual hydrodynamic fluid calculations require advanced math, we only use the basic, algebraic calculations in this program since we don't need high fidelity in our physics modeling. Also, there is a good chance that the results in this model aren't completely accurate; but the project should be sufficient to demonstrate how a liquid storage and transfer design should work.

There are also a number of things a programmer can do from here. The original text-based code could be changed to use a database instead of class instances; the rest of the GUI could be modeled out; there are always more tests to write, and so on. Hopefully, you have gained sufficient knowledge of Python to feel comfortable writing your own programs now.

Other Books You May Enjoy

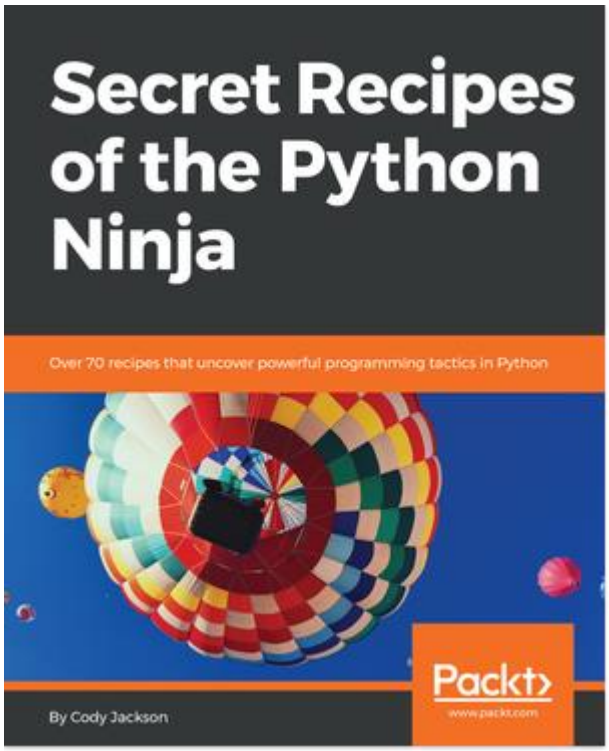
If you enjoyed this book, you may be interested in these other books by Packt:



Learn Python Programming - Second Edition
Fabrizio Romano

ISBN: 978-1-78899-666-2

- Get Python up and running on Windows, Mac, and Linux
- Explore fundamental concepts of coding using data structures and control flow
- Write elegant, reusable, and efficient code in any situation
- Understand when to use the functional or OOP approach
- Cover the basics of security and concurrent/asynchronous programming
- Create bulletproof, reliable software by writing tests
- Build a simple website in Django
- Fetch, clean, and manipulate data


Secret Recipes of the Python Ninja
Over 70 recipes that uncover powerful programming tactics in Python
By Cody Jackson
Packt
www.packt.com

Secret Recipes of the Python Ninja
Cody Jackson

ISBN: 978-1-78829-487-4

- Know the differences between .py and .pyc files
- Explore the different ways to install and upgrade Python packages
- Understand the working of the PyPI module that enhances built-in decorators
- See how coroutines are different from generators and how they can simulate multithreading
- Grasp how the decimal module improves floating point numbers and their operations
- Standardize sub interpreters to improve concurrency
- Discover Python’s built-in docstring analyzer